Kahn Process Networks as Concurrent Data Structures: Lock Freedom, Parallelism, Relaxation in Shared Memory

Nhat Minh Lê

To cite this version:

Nhat Minh Lê. Kahn Process Networks as Concurrent Data Structures: Lock Freedom, Parallelism, Relaxation in Shared Memory. Programming Languages [cs.PL]. Ecole normale supérieure - ENS PARIS, 2016. English. NNT : . tel-01684181

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à l'École normale supérieure

Kahn Process Networks as Concurrent Data Structures

Lock Freedom, Parallelism, Relaxation in Shared Memory

ED 386 SCIENCES MATHÉMATIQUES DE PARIS CENTRE
Spécialité INFORMATIQUE

COMPOSITION DU JURY :

M. COHEN Albert
Inria & ENS, Directeur de thèse

M. DODDS Mike
University of York, Rapporteur

M. DONALDSON Alastair
Imperial College London, Examinateur

M. HARRIS Tim
Oracle Labs, Examinateur

M. HERLIHY Maurice
Brown University, Rapporteur

M. ZAPPA NARDELLI Francesco
Inria & ENS, Examinateur

Soutenue par Nhat Minh LÊ
le 9 décembre 2016

Dirigée par Albert COHEN

Remerciements

J'aimerais remercier les gens qui, d'une manière ou d'une autre, ont contribué à l'aboutissement de cette thèse.

Tout d'abord, mon directeur de thèse, Albert Cohen, mes rapporteurs, Mike Dodds et Maurice Herlihy, ainsi que l'ensemble de mon jury.

Mon équipe, PARKAS, ensuite. En particulier Adrien Guatto, pour sa relecture approfondie du présent manuscrit et les nombreux échanges productifs durant ces années dans le bureau S10. Robin Morisset, également, pour ses remarques et ses contributions, sans lesquelles libkpn ne serait pas la même aujourd'hui.

Francesco Zappa Nardelli, pour les cours à l'X. Andi Drebes et Antoniu Pop, pour les discussions techniques sur le parallélisme. Guillaume Baudart, Ulysse Beaugnon, Tim Bourke, et tous ceux que j'ai eu le plaisir de côtoyer aux pauses café. L'équipe tout entière pour m'avoir fait une place en son sein.

Enfin, je tiens à remercier ma famille et mes amis pour leur soutien.

Résumé

La thèse porte sur les réseaux de Kahn, un modèle de concurrence simple et expressif proposé par Gilles Kahn dans les années 70, et leur implémentation sur des architectures modernes, multi-cœurs et à mémoire partagée. Dans un réseau de Kahn, le programmeur décrit un programme parallèle comme un ensemble de processus et de canaux communicants, chaque canal reliant exactement un processus producteur à un consommateur.

Nous nous concentrons ici sur les aspects algorithmiques et les choix de conception liés à l'implémentation, avec en vue deux paramètres clefs : les garanties non bloquantes (lock freedom) et la mémoire relâchée. Le développement d'algorithmes non bloquants efficaces s'inscrit dans une optique de gestion des ressources (importante pour les systèmes embarqués) et de garantie de performance sur les plateformes à ordonnancement irrégulier, telles que les machines virtuelles ou les processeurs graphiques programmables. Un travail complémentaire sur les modèles de mémoire relâchée vient compléter cette approche théorique par un prolongement plus pratique dans le monde des architectures à mémoire partagée contemporaines.

Nous présentons un nouvel algorithme non bloquant pour l'interprétation de réseaux de Kahn. Celui-ci est parallèle sur les accès disjoints : il permet à plusieurs processeurs (ou plusieurs threads) de travailler simultanément sur un même réseau de Kahn partagé, tout en exploitant le parallélisme inhérent aux processus indépendants. Il offre dans le même temps des garanties de progrès global non bloquant, c'est-à-dire en mémoire bornée et en présence de retards sur les processeurs. L'ensemble forme, à notre connaissance, le premier système complètement non bloquant de cette envergure. Il met en œuvre une palette cohérente de concepts et de techniques classiques de programmation non bloquante (recyclage de la mémoire, mises à jour complexes avec assistance, etc.), et incorpore des idées et optimisations spécifiques aux réseaux de Kahn. Nous discutons également d'une variante bloquante destinée au calcul haute performance, avec des résultats expérimentaux encourageants.

Abstract

In this thesis, we are interested in Kahn process networks, a simple yet expressive model of concurrency, and its parallel implementation on modern shared-memory architectures. Kahn process networks expose concurrency to the programmer through an arrangement of sequential processes and single-producer single-consumer channels.

The focus is on the implementation aspects. Of particular importance to our study are two parameters: lock freedom and relaxed memory. The development of fast and efficient lock-free algorithms ties into concerns of controlled resource consumption (important in embedded systems) and reliable performance on current and future platforms with unfair or skewed scheduling such as virtual machines and GPUs. Our work with relaxed memory models complements this more theoretical approach by offering a window into realistic shared-memory architectures.

We present a new lock-free algorithm for a Kahn process network interpreter. It is disjoint-access parallel: we allow multiple threads to work on the same shared Kahn process network, fully utilizing the parallelism exhibited by independent processes. It is non-blocking in that it guarantees global progress in bounded memory, even in the presence of (possibly infinite) delays affecting the executing threads. To our knowledge, it is the first lock-free system of this size, and integrates various well-known non-blocking techniques and concepts (e.g., safe memory reclamation, multi-word updates, assistance) with ideas and optimizations specific to the Kahn network setting. We also discuss a blocking variant of this algorithm, targetted at high-performance computing, with encouraging experimental results.

Contents

List of Figures

Chapter 1

Introduction

The contemporary parallel-programming landscape is quite vast, with its many languages, frameworks and libraries. At the lowest level lies the raw interface of machines: processors each running a sequence of instructions, communicating through shared memory and other devices. At a higher level, there are several competing paradigms that offer various tools and abstractions to the programmer, and their implementations, whose job is to translate these constructs into a raw low-level form involving threads and synchronization primitives native to the target architecture. Several factors need to be balanced in any such implementation. Do we want more performance? Do we need to satisfy particular system or hardware constraints? Latency? Throughput?

In this thesis, we are interested in a specific model of concurrency, namely, Kahn process networks, and its parallel implementation on modern shared-memory architectures. Kahn process networks expose concurrency to the programmer through a simple arrangement of sequential processes and single-producer single-consumer channels. In a sense, they share much in common with the popular task-based paradigm, mostly found in performance-oriented applications such as numerical computations [Buttari et al., 2009], and supported by mainstream solutions such as OpenMP. They also feature an innate notion of streaming, through their channels, which appears to be a promising direction for performance-minded parallelism. At the same time, the model has seen indirect uses in embedded systems in the form of synchronous data-flow languages such as Lustre [Caspi et al., 1987]. We believe these characteristics make for a good candidate to a new experimental implementation: well-understood and exhibiting known use cases, yet offering a little something more in the way of perspectives.

The focus is on the implementation aspects. We target a simple generic multi-threaded system with shared memory and very basic synchronization capabilities, which may equally represent a full-fledged multi-tasking operating system, or a bare-bone multicore machine. Of particular importance to our study are two parameters: lock freedom, and relaxed memory. The former dictates what can be expected of the low-level thread scheduling; the latter concerns synchronization through shared memory. Research into a reasonably fast and practical lock-free implementation ties both into concerns of controlled resource consumption (time and memory), which are linked to embedded-system uses, and reliable performance on current and future platforms with unfair or skewed scheduling (e.g., virtual machines, or GPUs). Our work with relaxed memory models complements this more theoretical approach by offering a window into realistic shared-memory architectures of today.

Our main contributions are:

- A lock-free disjoint-access-parallel algorithm for the interpretation of Kahn process networks, which allows multiple threads to work on and communicate with a shared Kahn process network. It is lock-free in the sense that threads are always able to assist each other in evaluating any Kahn process, such that, at any one time, no more than one executing thread is necessary to make progress (i.e., produce output from the process network), provided enough external input is available from the environment. In particular, delayed threads are able to rejoin the fray regardless of how long they have been asleep and what information they might have missed, including communications with the environment. It is parallel on disjoint accesses, meaning that threads working on disjoint parts of the Kahn process graph do not interfere.

  To the extent of our knowledge, the interpreter is the first lock-free object of its kind, both in terms of size and the abstraction it provides: we offer a non-blocking progress guarantee at the system level, while implementing higher-level blocking communication for Kahn processes themselves.

- A high-performance, but not lock-free, spin-off of our Kahn process network implementation. This version is fully implemented and tested as a C11 library. It comes with specially tailored parallelizations of a couple linear algebra algorithms, and experimental results, which demonstrate performance on par with—and sometimes better than—state-of-the-art specialized implementations such as Intel MKL and PLASMA.

- A complete linearizability study of the lock-free algorithm under standard sequentially consistent, i.e., interleaving, rules, as well as further analyses of select parts of both the lock-free and performance-oriented implementations in the C11 relaxed memory model. This includes detailed proofs of two basic algorithms: the Chase–Lev work-stealing deque, and a caching variation of a single-producer single-consumer ring buffer.

It is worth mentioning that our approach assumes very little about the host platform. In particular, we do not depend on automatic garbage collection. In the case of the lock-free algorithm, the code is explicitly integrated with a lock-free memory reclamation system (based on either reference counters or hazard pointers). As such, it is safe to use in an environment with finite memory (sufficient for the execution of the process networks of interest) and maintains strong resource-usage guarantees: it incurs no additional delays, and, in the worst case, only leaks a bounded amount of memory.

This dissertation is organized as follows. In Chapter 2, we explain the necessary technical background related to concurrent and lock-free programming techniques. We first describe basic assumptions about the implementation environment and review definitions of fundamental concepts such as linearizability and lock freedom. We then introduce several programming patterns. Chapters 3 and 4 present our contributions. Chapter 3 starts with a description of Kahn process networks, then dives into the main lock-free Kahn process network algorithm. We take a bottom-up approach, by building progressively larger linearizable components on top of each other. Finally, Chapter 4 deals with practical issues of process scheduling and relaxation in the C11 memory model (proof techniques and performance). It closes with a discussion of the performance-oriented blocking implementation and accompanying applications.

Chapter 2

Introduction to lock freedom

Before we delve into the Kahn-specific bits of our work, let us first reflect on the fundamental topic of non-blocking concurrent programming. To those new to it, this chapter may act as an introductory text; to others, it should serve to clarify what precise flavor of lock-free algorithms is developed in the rest of this work. Throughout, we aim to provide the reader with the same insight that has guided us through our journey into the world of lock freedom.

Sections 2.1 to 2.3 review the concurrent properties of the language our algorithms are written in, as well as essential concepts such as linearizability. Sections 2.4 and 2.5, describe some traditional generic lock-free techniques based on pointers and arrays, which form the foundations for the data structures presented in later chapters. Finally, in Sections 2.6 to 2.8, we address the question of explicit memory management, which is central in non-blocking systems such as ours, which do without automatic garbage collection.

2.1 Sequential consistency, lock freedom and linearizability

In this document, algorithms are written in a C-like imperative language, with integers and pointers (including basic pointer arithmetics), but excluding any non-local control structures (e.g., *setjmp*). We add a few convenience features—such as tuples and basic type inference—which are explained on first use and should be quite intuitive regardless. Sequential programs written in this language behave as would their C counterparts.

We showcase these different syntactic elements through the simple example in Figure 2.1 of a ring buffer. Notice the *var* keyword, under *pop*, used to declare variables whose type can be infered from the right-hand side of an assignment—this is similar to the *auto* keyword in C++11 or the *var* keyword in C#. Also, we repurpose the comma separator to declare, construct, and match tuples, as seen in the return type of the same function, akin to its role in the Go language.

We now consider concurrent programs made of several sequential **threads** of execution. As usual, each thread is simply a subprogram written in the sequential fragment. Threads may communicate only through **shared** memory. Global variables not declared with the *thread_local* specifier, as well as regions allocated through the system allocator

```
// At this stage, let us not concern ourselves with integer wrap-around.
int front ≔ 0;
int back ≔ 0;
int data[L];

bool push(int x)
{
    if (back − front = L − 1)
        return false;
    data[back % L] ≔ x;
    ++back;
    return true;
}

(bool, int) pop()
{
    if (back = front)
        return false, 0;
    var x ≔ data[front % L];
    ++front;
    return true, x;
}
```

Figure 2.1: Simple ring buffer

(e.g., *malloc*) are considered shared. Local variables declared within functions and global variables declared *thread_ local* are private and inaccessible to others. In particular, we never publich the address of a local variable to shared memory.

Threads are defined similarly to normal procedures, except with the additional *thread* specifier, which makes calling such a routine spawn a thread, as shown in Figure 2.2. Execution starts with a single thread calling the *main* function, as is customary in C. Most of the time, however, we simply assume already existing threads, when there is no risk of confusion. Each running thread has a thread identifier, which is an integer stored in the special read-only variable *threadid*. In certain algorithms, we further assume a bounded maximal number of threads *THREAD_ MAX*. Those are usually lower-level algorithms designed to synchronize between hardware processors, which normally come in fixed numbers.

The above notion of thread is intentionally very bare bone. Most importantly, threads start only as a result of being invoked explicitly from another piece of code, and control flow within each thread is purely sequential. There is no supervision, no clocks, no interrupts or other forced context switches, no signals or system-wide barriers.

We assume the semantics of a multi-threaded program is **sequentially consistent**, as originally defined by Lamport [1977]: it is given by interleaving. That is, execution may alternate between threads, one instruction at a time—control structures such as loops and conditionals are unrolled appropriately. It never stops midway, meaning instructions are **atomic**. And at any point in time, every changes to the shared memory made by previous statements, whether they are from the same thread or a different one, are immediately apparent to all further statements. Conversely, the order between operations that do not affect shared memory does not matter as far as the observable shared state is concerned. To sum up, any execution of the system is equivalent to a sequence of instructions taken from the various threads, which is compatible with the local orderings from the threads from which they originate.

In our simple buffer example, it was shown by Lamport that the code of Figure 2.1 does not need any particular adjustment to work under sequential consistency, and is thus identical to its sequential version. The only difference is in its usage: instead of a single sequence of push and pop operations, we have two distinct threads, the *producer* and the *consumer*. They take on exclusive roles, as one inserts new data into the queue, and the other retrieves it.

2.1.1 What is lock freedom?

In a sequentially consistent world, each possible execution of a concurrent program can be described by a specific interleaving, which is a sequence of scheduler choices, or **schedule**. More precisely, a schedule is a finite or infinite word over the set of threads (denoted by their identifiers). For example, going back to the Lamport queue, if we note P the producer thread and C the consumer thread, $CCCPCP$ represents the execution in which the consumer is allowed three steps, followed by one step from the producer, and again one step from the consumer and another by the producer. With every choice in a given schedule, we associate the action taken by the corresponding step in the chosen thread.

```
thread void producer()
{
    for (;;) {
        // Get some value to push.
        var x := input();
        while (not push(x))
            continue;
    }
}
```

```
thread void consumer()
{
    for (;;) {
        bool ok;
        int x;
        do
            ok, x := pop();
        while (not ok);
        // Do something with x.
        output(x);
    }
}
```

```
int main()
{
    producer();
    consumer();
    return 0;
}
```

Figure 2.2: Producer and consumer for the simple queue

This defines a total order over actions taken during an execution, named **happens-before**; more casually, we may say that an instruction happens (or occurs) before or after another.

Given an execution of a concurrent program, call and return actions in a same thread work strictly in tandem: we do not allow non-local control. Following the sequential semantics of the language, calls stack and are **matched** by return statements in a first-in last-out manner; at most, there may only be as many return actions as there are calls. A call may be unmatched if there are not enough return instructions in the given schedule. An **invocation** is the interval between a call and its matching return action, if it exists, or the end of the execution, otherwise. If the former holds, the invocation is said to be **complete**; otherwise, it is incomplete.

We consider functions that operate on a same set of shared locations, such that no other part of the program may access those locations except by calling or being called by those functions. By convention, we refer to this entire set of variables as a **concurrent object** (or simply, object), and the associated functions as its **methods**. Without loss of generality, we suppose that methods of a same object behave as if they do not call each other; internal helper procedures do not count as methods. The fact that only methods of a certain object may operate (perhaps indirectly) on its contents is known as **non-interference**, and is a prerequisite for the proper operation of every algorithm presented in this chapter and beyond.

Note that some objects can be part of larger objects, eventually making up the entire program, if we want. This simply provides us with a notion of scope to work with.

We further restrict our study to concurrent objects that contain no input or output actions (e.g., calls to *stdio* functions), and whose sole means of communication is therefore shared memory.

An object is **non-blocking**, or **lock-free**, if: for every permissible schedule γ, if γ is infinite and contains an unmatched invocation of some object method, then γ contains an infinite number of complete method invocations.

If we break down the definition into alternatives, equivalently, γ must verify at least one of the following propositions:

- γ is finite;
- γ contains only complete invocations of the object methods;
- or γ contains an infinite number of complete invocations of the object methods.

Intuitively, a non-blocking object always guarantees that if its methods run long enough (γ does not end abruptly in the middle of some invocations), some instance will eventually return. In other words, lock freedom ensures progress (in the form of method completion), even in the presence of **arbitrarily unfair scheduling choices**, reflected in the quantification over all possible schedules.

As a side remark, according to this, all programs that eventually make progress in a bounded number of steps from any configuration are considered lock-free. This includes algorithms that do not exhibit a constant, state-independent, limit on such bounds.

Note also that if we removed the third alternative from the above list, we would require that all invocations complete. This stronger condition is known as **wait freedom**.

In the previous queue example, neither the *push* nor the *pop* methods contains any kind of looping construct, so the procedures cannot block. Therefore, Lamport's queue is lock-free (actually, wait-free).

It is useful, however, to pause for an instant and ask ourselves what "lock-free" means, precisely, for a single-producer single-consumer ring buffer. What about the cases where the queue is full or empty, thus triggering a false return from either operation? To begin with, the two threads perform asymmetrical work: one produces values while the other one consumes them. This brings about a fundamental, and general, point: can a system that performs heterogeneous computations on its different threads ever be non-blocking? Will the failure of one thread not inevitably deprive the whole of one essential part of the calculation? In the case of our producer–consumer pair, if either one halts, the other is left hanging with either an empty or a full buffer.

Yet, looking at the sample code in Figure 2.1 and our previous definitions, all conditions appear to be satisfied. This apparent contradiction arises because of an important notion, which is clearly illustrated on this simple example: scope. In other words, lock freedom depends on what object and methods we consider. We have chosen to make both *push* and *pop* return, regardless of whether they succeed in adding or removing an element at all. Thus, there is a bounded number of steps between the beginning and the end of any invocation, making the system wait-free, hence lock-free.

While all this might seem bizarre at first, we should remember that the object of interest is the shared ring buffer, not the computation as a whole. And indeed, the *push* and *pop* methods are non-blocking with respect to each other, and for the progress criterion chosen, which accounts for failures due to full or empty buffers, which are treated as out of scope, and thus discounted.

If instead we write the *pop* function as in Figure 2.3, with a loop instead of the conditional, on line 3, then, according to our definition, this object is blocking, since the consumer can starve in the loop, waiting for a halted producer. We can, however, agree that both versions could be considered single-producer single-consumer queues, albeit with different interfaces. In general, lock freedom is a relative criterion, and it is up to the us, the programmers, to come up with reasonable and useful specifications for our non-blocking objects and methods.

2.1.2 Object specifications

In the above definition of lock freedom, we quantify over all permissible executions. This brings the question of how a concurrent object ought to be used by some client code. Concretely, each object has an associated predicate that filters allowed executions. Naturally, adding more restrictions to the filter gives us more hypotheses to work with, but also makes life harder for clients. Depending on the kind of properties that we wish to describe, different methods exist.

In our single-producer single-consumer queue example, the contract deals with ownership and exclusion: there may only be a single consumer and a single producer at any

```
int pop()
{
   while (back = front)
      continue;
   var x := data[front % L];
   ++front;
   return x;
}
```

Figure 2.3: Blocking *pop*

one time. In other words, the client must ensure that invocations of each method are non-overlapping and totally ordered. In a more formal setting, concurrent program logics offer tools to express this kind of conditions accurately. For example, any framework descended from the concurrent separation logic of O'Hearn [2007] should be able to handle such constraints quite easily, through the use of some form of permissions or another.

For many concurrent objects, the protocol imposed on clients rarely exceeds such simple exclusion property. In fact, a large number of data structures, including most specifications of dynamic sets such as lists or maps, allow their methods to be called in any order, anytime. For our purpose, we keep to simple informally written predicates to describe client obligations.

Aside from requirements on the part of the client, an object also usually offers guarantees about the expected behavior of the data structure interface when used correctly. It might seem strange to ask ourselves how to express what our code does, when the code itself already has an attached semantics. However, given the complex nature of concurrent algorithms and the multitude of implementations, it should come as no surprise that people have come up with more abstract ways to specify their behaviors.

2.1.3 Linearizability

In sequential consistency, undeniably, the most popular abstraction criterion is linearizability, as defined by Herlihy and Wing [1987]. The ultimate goal is to relate the semantics of a concrete concurrent implementation to a sequential specification where method invocations from the client occur exclusively one after another—as if the methods themselves never call each other.

Therefore, we start with a sequential version of our data structure, that exposes the same methods. What is important here is not how it works, but rather what can be observed from the outside, by calling said methods and reading their return values. Notice that, again, we do not allow or consider more advanced sequential control structures, such as non-local exits.

There are several ways to formalize these observations: any relation that describes allowed (or forbidden) call and return sequences constitutes a valid specification. For example, we could simply state: the single-producer single-consumer queue allows *pop* to successfully return a value only after matching a *push* call.

Linearizability, however, offers a more systematic approach. Informally, linearizability, with respect to a given sequential specification, is the property of a concurrent object to behave as if calls to its methods were performed atomically in a sequence compatible with said specification. There are two commonly found, equivalent, definitions of linearizability.

2.1.3.1 First definition of linearizability

The first characterization is adapted directly from Herlihy and Wing [1987] with only a few alterations to suit the above notion of execution.

We start with a few definitions. A **history** is a sequence of method call and return actions. Each such action is labeled with a thread identifier, an object, method name, arguments and return values. Those call and return actions can be matched or unmatched and make up invocations as defined in the previous subsection. Again, same-object method invocations do not nest sequentially. Given a history H, we note complete(H) its restriction to actions that form complete invocations, i.e., matching call and return actions.

A history is **sequential** if it is a sequence of consecutive matching pairs of call—return actions, except for the last call, which may be unmatched. A sequential specification is a set of legal sequential histories. If a history is not sequential, it is **concurrent**. We can extract a concurrent history for a given set of objects from any execution.

We say that a concurrent object is **linearizable** to a sequential specification if, for every permissible execution, the extracted history H can be extended to (is a prefix of) some history H' such that:

- complete(H') is a permutation of some legal sequential history S;
- and if a return action precedes a call action in H', it also does in S.

Let us take a step back to look at what the sequential history S, which we call the **linearization** of H, means for the client, and how it can be considered equivalent.

The extension to H' and restriction to complete(H) work in tandem. They govern what we do with unmatched calls in the original execution. Although the completion of those invocations has yet to be observed (they have yet to return), the effects of some of them may already be observable through shared memory. Therefore, we allow the extended history H' to treat them as if they had completed. Conversely, we want to treat those invocations that have not taken effect yet as if they had not started at all, hence complete(H').

Along with the fist condition listed, this tells us that no matter what actually happens inside the procedures, our implementation behaves as if it were an interleaving not of singular instructions, but of whole method invocations.

The second requirement can be justified by acknowledging that the client code may enforce a specific ordering between invocations. While it has no control over the control flow when execution is inside a method, it may very well constrain a specific instance to finish (return) before another begins (call).

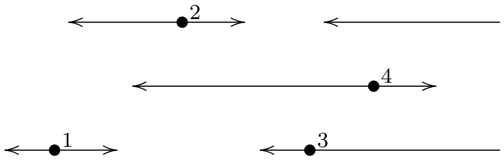


Figure 2.4: Linearization points

Note that shuffling around actions from a same thread is forbidden by virtue of their already being perfectly alternating call–return pairs. They therefore enforce a local return-before-call ordering between all the pairs. Thus, locally, each thread observes the same sequence of calls and returns in both the sequential and concurrent histories. Those calls are then reordered on the global timeline according to the rules above to produce a meaningful succession of states that can be explained by the sequential specification.

2.1.3.2 Second definition of linearizability

There is another common way to define linearizability, introduced as a lemma in [Herlihy and Wing, 1987]. We remark that to get a sequential history, first, it is necessary to group all matching call and return actions to form consecutive pairs. Then, those pairs are moved around under the constraint that returns that precede calls in the original history remain in that order in the permuted result. If we represent time as a line and method invocations as segments on that line, then each segment is transformed into a dot representing the resulting pair. The return-before-call rule now states that non-overlapping segments stay in the same order. Equivalently, the two statements taken together can be summed up by choosing, for each segment, one point to represent it from the segment itself. Every complete invocation must include one such point; incomplete invocations may or may not do so (depending on whether they are appended by H' or excluded by complete(H')). The order of the points is the linearization.

Figure 2.4 illustrates this approach. For every complete invocation segment as well as one incomplete segment, we have chosen a point. The order of those points (denoted in superscript) yields a sequential history—which still needs to be proven correct with respect to the specification. Notice that the third call has yet to return (the arrow does not finish on the right), yet linearizes before the fourth, whose return has been observed already, while the upper-right unmatched call does not have an associated point and is totally absent from the linearization.

It is easy to see that this arrangement meets all the conditions, in particular, since a point on a segment never gets placed after another on a disjoint following segment. To convince ourselves that the other way around also works, consider a linearization obtained under the above definition. We begin with the segments from the concurrent history. Suppose actions (segment ends) lies on unit positions and no two actions share a position (since they are strictly ordered according to sequential consistency). We note segments $[a_i; b_i[$, where the subscript i follows the order of the corresponding invocations in the sequential history.

For the first pair of the sequence, we pick its representative point from the corresponding segment $[a_0; b_0[$. Let us call $x_0 = a_0$ its position on the line.

Suppose that at rank n, points $x_0, \ldots, x_{n-1}$ have been assigned from the sequential history, that they all lie on their respective segments, and that one of those segments, $[a_i; b_i[$, is such that $x_{n-1} - a_i \leq \sum_{k=1}^{n} \frac{1}{2^k}$ with x_{n-1} not an integer.

For the following pair, with segment $[a_n; b_n[$, we choose the point whose position is $x_n = \max(x_{n-1} + \frac{1}{2^{n+1}}, a_n)$. If $x_{n-1} < a_n$, the point lies exactly at the beginning of the segment. Otherwise, there exists a previous segment whose lower end a_i satisfies $x_{n-1} - a_i \leq \sum_{k=1}^{n} \frac{1}{2^k}$ (by induction) and $a_i \neq b_n$ (by definition). Since pair i in the sequential history appears before pair n, it must be that segment i does not start after segment n; else, they would be disjoint, and their order would have to be preserved by the reordering. Thus, $a_i < b_n$, and $a_i < x_{n-1} + \frac{1}{2^{n+1}} \leq x_n = a_i + \sum_{k=1}^{n+1} \frac{1}{2^k} < a_i + 1 \leq b_n$. The new point lies on its segment. We verify that $x_n \leq a_i + \sum_{k=1}^{n+1} \frac{1}{2^k}$.

We have thus shown that a possible way to create compatible sequential histories is to assign to each invocation (at most) one point in time that lies its lifetime—between its beginning call and return instructions. This construction actually provides a second interpretation of linearizability: an object is linearizable if all its methods have an **effect point** (or **linearization point**) during each invocation where they can be considered to occur atomically, and the sequence of such points is compatible with the sequential specification.

For most purposes, this characterization is both easier to use[1] and simpler to understand in practice: at a given instant, the effects of any invocation whose linearization point comes before are fully seen; any point that comes after is not yet visible. It does obscure, however, some aspects of the reasoning about which orderings in the original concurrent history are important and must stay intact, and which can be broken or modified.

2.1.4 Linearizability of the simple queue

As an example, let us reexamine our single-producer single-consumer queue for clues about linearizability. As is often the case with basic data structures, the sequential semantics are directly provided by the sequential ring buffer. Its abstract state is therefore a list of current values v, to which we add two (unbounded) integers f and b that keep track of how many successful invocations (that return true) to *pop* and *push* respectively have been registered. The concrete state is, of course, made of the three variables *front*, *back*, and *data*.

In the *push* method, new data is only visible once the new *back* index has been written. Symmetrically, in *pop*, the consumption is actually signaled when *front* is increased. In failing invocations, the linearization point can be taken as the first test, where the

[1] Using the effect points to enact proofs of linearizability only works reasonably if the points are mostly constant for each method; if every invocation requires a different linearization point, it essentially boils down to direct construction of histories.

opposite index is checked.[2] Although they do not contribute any changes to the data structure, a suitable instruction must be chosen in accordance with other effect points so as to build a permissible sequential history.

We now consider a modified abstract machine that keeps track of the abstract state as well as the normal shared variables. The concrete state of the program is updated by each instruction, according to the usual semantics. The abstract state, on the other hand, only changes at linearization points. If the attempted operation is not valid according to the specification, the machine halts with a simulation error.

Our goal is to show that given any concurrent history valid on the normal machine, the same history can be reproduced on the augmented machine without simulation error. To that end, we define an invariant over both concrete and abstract states:

$$\begin{gathered}0 \leq \text{front} < L \land f \equiv \text{front} \mod L \land \\ 0 \leq \text{back} < L \land b \equiv \text{back} \mod L \land \\ 0 \leq b - f < L \land \\ \text{data}[f \ldots b-1] = v\end{gathered}$$

Where $\text{data}[f \ldots b-1]$ is defined as a word over the element type T such that:

$$\begin{array}{rl}\text{data[front]} \ldots \text{data}[\text{back}-1] & \text{if front} \leq \text{back} \\ \text{data[front]} \ldots \text{data}[L-1]\text{data}[0] \ldots \text{data}[\text{back}-1] & \text{if front} > \text{back}\end{array}$$

Notice that $|v| = b - f$ and v is empty when $b = f$ and full when $b = f + L$. By definition, b is incremented by one when a successful *push* linearizes; similarly for f on a successful *pop*. Trivially, both b and f monotonically increase.

Furthermore, we recall that invocations of *push* are exclusive; same for *pop*. Therefore, between two instructions of *push* (resp. *pop*) only statements from *pop* (resp. *push*) may occur. Thus, b (resp. f) is invariable in between *push* (resp. *pop*) statements. The same argument can be made about concrete variables.

It remains to be shown that every statement maintains the invariant. In particular, we need to prove that given the invariant on input, linearization points both preserve it and allow for a sensible operation on the abstract value.

- Local computation and control statements (e.g., call, return) do not modify the global state. Neither do read instructions.

- However, the first test in either method is a linearization point for failed invocations, which return false. According to the invariant, v is empty exactly if $b = f$, hence back = front, and full exactly if $b = f + L$, hence $back - \text{front} \equiv L - 1 \mod L$. Thus the return actions follow the specification.

- The stores to *data* and *back* are part of a successful invocation of *push*, as depicted in Figure 2.5. We note x_i the value of variable x right before line i.

[2] Note that the order in which the variables are read is not significant as one of them is exclusively written by the current thread.

```
1 **assert** back − front $\neq$ L − 1;
2 data[back % L] := $x$;
3 ++back;
4 **return** true;
```

Figure 2.5: Events in a successful invocation of *push*

```
1 **assert** back $\neq$ front;
2 $x$ := data[front % L];
3 ++front;
4 **return** true, $x$;
```

Figure 2.6: Events in a successful invocation of *pop*

- On line 1, we have $b_2 = b_1$ and $f_2 \geq f_1$. Therefore, $b_2 - f_2 \leq b_1 - f_1 < L - 1$ and the store at index *back* does not override any value present in v_2.
- On line 3, we have data[back$_3$] = data[back$_2$] = x and, as above, $b_3 - f_3 \leq b_1 - f_1 < L - 1$. After the increment to *back*, $b - f = b_3 + 1 - f_3 \leq b_1 + 1 - f_1 < L$, and the sequence data[$f \ldots b - 1$] is extended with the new element at data[back$_3$] = x. At the same time, the abstract value registers the effect of push(x) and becomes vx, which matches.

- Similarly, the load from *data* and store to *front* are part of a successful invocation of *pop*, as seen in Figure 2.6.
  - On line 2, we have $f_2 = f_1$ and $b_2 \geq b_1$. Therefore, $b_2 - f_2 > 0$ and data[front] is the first element of data[$f \ldots b - 1$], hence the first element of v.
  - On line 3, we still have $f_3 = f_1$ and $b_3 \geq b_1$, hence $b_3 - f_3 > 0$. After incrementing *front*, the sequence data[$f \ldots b - 1$] loses its first element, as does the abstract value v upon registering the successful *pop*.
  - On line 4, we finally return the value read earlier, which matches the expected semantics of *pop* effected on line 3.

We conclude that the invariant is preserved, v has a legal value at all times, and both methods return expected results with regard to the abstract value at their point of effect. Thus, our simple queue is linearizable to a sequential bounded ring buffer.

2.1.5 Breaking down the linearizability of the simple queue

The fundamental strategy behind the proof above is the superposition of an abstract object on top of the implementation, similar to what is used by Herlihy and Wing [1987]. As usual in algorithmic reasoning (usually under the guise of *invariant strengthening* or similar terms), the added variables must not only account for the desired semantics, but also any intermediate properties necessary to the invariant. In the above development, we do not strictly need the f and b abstract indices to describe the sequential queue, but they help write a stricter invariant ($b - f < L$) than would have otherwise been possible with just *front* and *back*.

The rest of the technical analyses follows methods borrowed more from the realm of weak memory models. Namely, we construct action structures[3]: sequences of instructions that match some specific scenario (e.g., a successful invocation of *push*). These let us derive arguments by exploiting local information (e.g., from prior branches).

As can be seen through this example, proofs of correctness are often quite involved, even for very simple algorithms.[4] Other data structures visited in this chapter are not systematically examined. We mostly keep to simple semi-formal arguments about states and sequences, where appropriate, and instead concentrate on more specific questions regarding the design and implementation of lock-free systems.

2.2 Reading from and writing to shared memory

In the previous section, we have worked with the usual imperative instruction set. In particular, memory was accessed as we would have normal variables in a sequential program. We now make explicit three additional constraints that will become necessary in our future analyses:

1. each statement may only contain one reference to some shared object;

2. if the reference is on the left-hand side of an assignment, then it is the entire left-hand side (i.e., it is not merely used to compute the address where the value should be written).

An assignment with a shared left-hand side is a **store** instruction; otherwise, the reference is read and corresponds to a **load** instruction.

In effect, we are disallowing—for now—complex atomic operations that affect multiple memory locations at the same time. This essentially leaves just instructions that map to register–memory operations in hardware.

While those simple load and store instructions are sufficient to build basic concurrent data structures with a limited notion of lock freedom—as in the case of Lamport's simple queue, many non-blocking algorithms require more powerful shared-memory primitives, such as the ones we introduce below.

2.2.1 Limitations of load and store instructions

Simple load and store instructions are insufficient for most lock-free algorithms. At the core of the problem lies the notion of **atomicity**: a statement is atomic if it is either fully

[3] As is frequently done for recent axiomatic memory models such as C11, for example in Batty et al. [2013].

[4] Although some, such as O'Hearn et al. [2010], have noted that sometimes more complex algorithms do not necessarily translate into more complex proofs of linearizability.

executed, or not at all. According to the sequentially consistent model described above, all statements are always atomic: the full value of a variable is read on a load, and its entire location is overwritten on a store. We could imagine other execution hypotheses, where objects could be read or written partially (in groups of lower–higher bits, for example), but we postpone such discussions, for the time being, to concentrate on a larger problem.

Even so, this guarantee is usually too weak. It is often the case that we want to read a variable in order to update its value. Take, for example, the simple counter: an object that linearizes to an integer with a single *increment* operation, whose purpose is to add one to its value and return the previously stored one (see Figure 2.7). The use cases are many: assigning an order of passage to some clients, counting occurrences of some events, etc.

The code in Figure 2.7 performs fine in a sequential environment; however, if we tried to run it from multiple threads, we might notice that values would be repeated from time to time. This anomaly stems from the fact that the first two instructions within *increment*, when taken together, are not atomic. Therefore, two concurrent threads could start both reading the same value *n*, only to both assign the same value back to *counter*, hence a duplicated return value.

Unfortunately, there is no way to write such a counter using only load and store instructions while retaining the desirable non-blocking property.

2.2.2 There is no non-blocking load–store counter

It is known since Herlihy [1988] that a wait-free counter using only load and store instructions does not exist. Below, we provide a retelling of this story, featuring a lock-free counter in our imperative model. Doing so also serves to introduce some proof techniques that will come in handy later on.

Imagine we have a counter data structure, written in our simple concurrent language. It has some shared variables, starting in a well-known initial state, and exposes a single method: *increment*. From there, suppose that we are in total control of the scheduler and want to find issues with the algorithm. Let us show that if it satisfies the above counter interface and uses only load and store instructions, then it cannot be lock-free.

Note that the code can be as sophisticated as we want. However, as stated in Section 2.1.1, we disallow explicit input and output from within concurrent objects. In this case, since *increment* receives no arguments, it does not have access to any source of information aside from what it has itself written to shared memory from previous invocations. In particular, it cannot rely on external oracles. This is important, in general, in lock-free programming: if there exists some mechanism that allows us to observe (or better yet, control) the scheduler itself, then different, often more lightweight, strategies are possible. Those techniques are mostly outside the scope of this chapter, as they deal with an entirely different fault model; however, for some examples of what can be done in the presence of stronger scheduler constraints, see Section 2.6.1.

That being said, our goal is now to craft a scheduling sequence involving concurrent calls to *increment* such that either two or more instances return the same value (which violates the interface contract), or the sequence is infinite without any call returning

```
// Again, we do not bother with wrap-around.
int counter := 0;

// Sequential semantics of the counter.
int increment()
{
    int n := counter;
    counter := n + 1;
    return n;
}

thread void counting()
{
    for (;;) {
        // Do something interesting.
        ...
        int x := counter();
        // We want to count the number of times something interesting
        // was done, with a margin of error up to $F$, where $F$ is
        // the number of thread faults.
        output(x);
    }
}
```

Figure 2.7: Sequential counter

(which violates lock freedom). In order to linearize to a counter, the code must satisfy some basic semantic properties:[5]

- from the initial state, if a thread executes a call to *increment* uninterrupted (it is the only thread schedule throughout the invocation), then the method returns the initial value (0);
- from the initial state, if two threads both call *increment* concurrently, one ends up acquiring 0 and the other one gets 1.

We consider a system with two threads T and T' and an initial sequence of instructions I as they are scheduled from either thread. For example, $I = TTT'$ would mean: evaluate two statements from T followed by one from T'. Throughout the proof, we strive to maintain the invariant that in state I (i.e., after executing the instructions from I), any single thread that runs uninterrupted should have *increment* return the same value $f(I)$. We start with $I = \epsilon$, the empty sequence.

We now define the integer $c(T, T', I)$, as follows. Informally, $c(T, T', I)$ is the number of consecutive steps that can be taken by T in the state I without acquiring $f(I)$.

More formally, consider the following setup. After the initial scheduling sequence I, we first let T run while T' sleeps, until it returns from its first call to *increment*, with value $f(I)$, after executing n statements. We then carry out the experiment again, except, this time, we stop after $n-1$ instructions in T, and switch to T' until *increment* returns in T'. Either the call in T' returns $f(I)$ or it returns $\neg f(I)$. If it returns $f(I)$, we can repeat this process again and again until we determine at which step $c(T, T', I) < n$ we assign $f(I)$ to T' instead of T. We can therefore define $c(T, T', I)$ as the highest value such that $IT^{c(T,T',I)}T'$ yields $f(I)$ for T'.

Note that $c(T, T', I) \geq 0$ by definition of I, and $c(T, T', I) = n$ is impossible, else T could return from *increment* without interference (because of lock freedom), thus netting $f(I)$. This would result in a duplicate value if followed by T' also returning $f(I)$.

Statement $c(T, T', I)$ in T, counting from zero after the initial sequence I, must be a store instruction to a shared variable. Otherwise, we could have stopped after $c(T, T', I) + 1$, as the shared state at $c(T, T', I)$ and $c(T, T', I) + 1$ would be indistinguishable.

We now enumerate the different scenarios governing the value of $c(T, T', I)$ and $c(T', T, I)$. We show that in all cases, it is possible to construct a longer prefix I that brings us back to our initial hypothesis, thus creating an infinite schedule, which contradicts lock freedom.

- If $c(T, T', I) \neq 0$, then the execution $IT^{c(T,T',I)}$ satisfies our invariant: whichever of T or T' runs first after that segment gets $f(I)$.
- Symmetrically, if $c(T', T, I) \neq 0$, we can take the sequence $IT'^{c(T,T',I)}$.
- Suppose both $c(T, T', I) = 0$ and $c(T', T, I) = 0$, meaning whoever starts running always gets assigned $f(I)$. This implies that the first statements after I in both T

[5] This is in fact a reformulation of the classic two-thread consensus problem by Herlihy.

and T' are store instructions. Either they both write to the same location, or they write to different shared variables.

- If they target different locations, then it makes no difference whether T or T' runs first. Executing $ITT'T\ldots$ is the same as $IT'T\ldots$ and $IT'TT'\ldots$ is the same as $ITT'\ldots$. Therefore, we can append TT' to I and start over with a reversed scenario, where the first to run gets the complementary value $\neg f(I)$.
- If they target the same location, then ITT' is equivalent to IT' as the value gets immediately overwritten. This in turn implies $c(T, T', I) \geq 1$, which contradicts $c(T, T', I) = 0$.

In all cases, we can extend I by a strictly positive number of instructions, and repeat the process indefinitely. Thus, we have built an infinite scheduling sequence that sees neither thread returning from its method invocation. It is what we call a **livelock** and precludes lock freedom, as no thread in the system makes any progress.

2.2.3 Explaining the impossibility of a load–store counter

The above proof might appear abstract and daunting at first, so let us take a couple of paragraphs to explain where the intuition lies, and look at some alternative algorithms. Starting from the conclusion, we have a statement of incompatibility. It is not possible to combine three criteria:

1. upholding the counting semantics;
2. having only load and store instructions in our arsenal;
3. while remaining non-blocking.

2.2.3.1 Counter semantics

If the first condition can be relaxed, it is possible to devise lock-free algorithms for some simpler tasks. For example, we could imagine partitioning the integers into subsets, one for each thread. This would work for assigning unique identifiers (e.g., by giving all even numbers to T and odd numbers to T'), but would not serve the purpose of counting, as an unbalanced work load or scheduling (e.g., one thread is consistently faster than the others) would produce glaring results, with high and low values, and many unclaimed positions in between. Following the reasoning of the previous subsection, we notice that this alternative algorithm does not satisfy the most basic assumption, that in the initial state ($I = \epsilon$), running either thread yields the same value $f(I) = 0$. Thus, the proof does not prooceed—it does not, in fact, even start.

We should not be hasty in dismissing this mechanism as mere sleight of hand. We could be tempted to exclaim: "Of course, if threads do not access the same memory locations, then everything is lock-free!" This is indeed the nature of such an algorithm. However, it is an important idea. Generally speaking, whenever we have only load and store operations at our disposal, we want to create a **partition** of memory, such that each thread gets assigned its own disjoint area to oversee. Unfortunately, as we will soon realize more fully, this comes at cost of much wasted memory, especially should a scheduling fault occur.

2.2.3.2 Only load and store instructions

The second constraint can be improved upon by including more powerful primitives, as we will see in the rest of this chapter.

But for the time being, let us first observe that it is impossible to write such an algorithm without any load instruction. Intuitively, if *increment* reads no value from shared memory, then it will always return the same sequence. This corresponds to the $c(T, T', I) = 0$ and $c(T', T, I) = 0$ case of the proof, where both sequences start with a store instruction. Essentially, if these are the only statements in our method, either it never returns (first subcase), or it will end with duplicate values (second subcase), which matches our intuitive prediction.

If load instructions are included, then it actually makes the proof easier, as illustrated by the first two branches, when $c(T, T', I) \neq 0$ or $c(T', T, I) \neq 0$. Since load statements do not modify the shared state, an adversarial scheduler can always choose to schedule those without affecting the outcome of any concurrent programs that might run just after them. Those programs have no way of even knowing whether any pure read instruction has been occurred.

But, concretely, what would happen in an actual algorithm? In practice, the concurrent program needs to account for the possibility of another thread having run in the time between its last instruction and the one it is about to execute. We could see it as a kind of election where the participants are threads. The threads attempt to elect who is going to get the right to some resource (here, the value $f(I)$). They exchange votes by means of store (casting a vote) and load (acknowledging a vote) instructions. In a lock-free algorithm, indefinite waiting is prohibited, as if the waiting thread happens to also be the last one alive, no progress is made. Thus even if both candidates agree on who gets $f(I)$ first, they can only ever strike conditional contracts. T might agree to let T' have the option of claiming $f(I)$ until a certain date, measured in terms of instructions, so as to satisfy lock freedom requirements. An evil omniscient scheduler can easily see through such a scheme and always choose to run T until its patience is exhausted, so that nobody can ever successfully exercise its option on the shared resource. Ironically, doing so leads to our threads failing to agree, and thus negates any lock-free claim we might initially have had.

2.2.3.3 Lock freedom

Indeed, abandoning lock freedom altogether leads to some interesting (blocking) algorithms. Even though this introduction expressly focuses on lock freedom, it does not hurt to examine what it is we lose and gain by temporarily lifting this restriction.

```
// We still do not bother with wrap-around.
int counter := 0;
bool flags[2] := {0};
int turn := 0;

int increment()
{
    flags[threadid] := true;
    turn := not threadid;
    bool f;
    int t;
    do {
        f := flags[not threadid];
        t := turn;
    } while (f and t = not threadid);

    int n := counter;
    counter := n + 1;

    flags[threadid] := false;
    return n;
}
```

Figure 2.8: Counter with Peterson's lock

Fundamentally, lock-based programming is all about waiting, while lock freedom is the opposite. In a primitive sense, a **lock** is essentially any device that allows one or more threads to wait for another. It is well-known that such objects can be implemented using only load and store instructions [Dijkstra, 1965, Peterson, 1981].

Figure 2.8 shows a counter based on Peterson's algorithm, for two threads. It works by having each thread signal its desire to acquire exclusive use of the *counter* variable by setting the corresponding slot in *flags*. A problem arises when both participants have declared interest; in that case, a decision needs to be reached, with one thread yielding to the other. This is reflected in the use of the write-shared *turn* variable, which can only hold a single value at a time: the identity of the winner. By the time both threads reach the while loop, both *flags* should be set, but *turn* is either zero or one, so exactly one loop will exit first into the exclusive middle portion of the code, called the **critical section**.

The system is not lock-free. If the scheduler decides to fault a thread within its critical section, then its *alter ego* gets stuck in the while loop forever, as *flags* is never again updated. Waiting in a loop such as the one in Peterson's algorithm is usually termed **busy waiting**, as the thread makes no significant action during this time; the

```
bool flags[2] := {0};

int reserve()
{
    for (;;) {
        // First test.
        bool f := flags[not threadid];
        if (f)
            return 1;
        // Set.
        flags[threadid] := true;
        // Second test.
        f := flags[not threadid];
        if (not f)
            return 0;
        // Reset.
        flags[threadid] := false;
    }
}
```

Figure 2.9: Bit reservation

loop is by extension known as a **busy loop**.

Another approach, as hinted at through our discussion of the load–store restriction above, is to have some kind of repeated election until an agreement is reached. Figure 2.9 illustrates this concept. It describes a system whose purpose is to assign distinct numbers 0 or 1 to two different threads. The algorithm uses only two shared locations, in the form of a *flags* array. A set flag signifies intent to take the lower value 0 by the corresponding thread; if either thread completes a full iteration of the loop uninterrupted, then the routines end.

To see why, let us consider the different executions of the two threads. First, we observe that if two consecutive tests agree (they read the same value), then the method returns. This means that if a thread is scheduled consistently long enough to perform two consecutive tests, then its program terminates, which in turn implies the opposite thread will get to run uninterrupted and also finish.

It remains to be proven that if they do, then the return values are distinct. The algorithm works by guaranteeing that the two invocations of *reserve* never take the same return path:

- Suppose both calls return 1. Then it must be that both *first tests* read true. However, one of the *first tests* must run first, and the one that follows will also follow the preceding *reset*. Since both tests are final—they lead to returns, no further instruction changes the value of the flag. Thus, the second *first test* cannot read true. Contradiction.

- Similarly, suppose both calls return 0. Then it must be that both *second tests* read false, yet both are also ordered. As before, the one that runs second follows the *set* of the opposite thread and cannot read false.

This kind of reasoning appears in many concurrent algorithms, and relies on two interleaved pairs of two instructions each: store S followed by load L. Each load statement reads from the opposite store location. Suppose threads T_0 and T_1 execute S_0L_0 and S_1L_1, respectively. Then there are only six possible scenarios, all of which start with either S_0 or S_1. If one starts with S_0 then L_1 will occur after S_0 thus cannot read a value older than the one written by S_0; symmetrically for S_1 and L_0. This assemblage is known as the **anti-store-buffering pattern** and although its use might seem obscure at this point—the proposed technique can often be avoided entirely in a sequentially consistent context through an appropriate choice of invariants—we will see further along that it becomes rather central in more relaxed memory models.

As we have mentioned, the present bit reservation routines only terminate subject to the condition that one thread gets to run without contention long enough. The class of algorithms that satisfy this criterion is known as **obstruction-free** [Herlihy et al., 2003]; as with wait freedom, precise study of obstruction-free algorithms is beyond the scope of this document.

2.3 Read-modify-write atomic instructions

As noted above, the only way to keep both lock freedom and the exact counter semantics is to relax the only remaining criterion: the restriction to load and store instructions.

The litterature has many examples of more complex primitive operations, but at the most basic level, they all share a common structure: they consist in an atomic grouping of a few simpler instructions. They are also called **read-modify-write instructions**, owing to the most oft-encountered shape, which has three stages: first, a shared variable is read, then, an operation is applied (which may involve local data and conditionals), then the variable is updated with the new value.

A simple example of such a construct is fetch-and-add, which linearizes to a sequential counter incrementation. In our model, it is equivalent to having the function depicted in Figure 2.10 execute atomically.

However, in practice, fetch-and-add is rarely seen outside of specialized algorithms. The reason for this lack of popularity is mostly the result of its being overshadowed by another compound operation by the name of **compare-and-swap** (also known as compare-and-set or compare-exchange). Both are readily available on modern hardware, but, as noted by Herlihy [1988], fetch-and-add is strictly less powerful than compare-and-swap, meaning some non-blocking algorithms cannot be written with fetch-and-add that can be written with compare-and-swap.[6] Figure 2.11 describes the semantics of the latter.

[6] Proof of this statement is beyond the scope of this introduction.

```
int faa(int ∗ptr, int c)
{
    int x := ∗ptr;
    ∗ptr := x + c;
    return x + c;
}
```

Figure 2.10: Fetch-and-add

```
// Writes new into ptr only if its current value matches expected.
bool cas(int ∗ptr, int expected, int new)
{
    if (∗ptr ≠ expected)
        return false;
    ∗ptr := new;
    return true;
}
```

Figure 2.11: Compare-and-swap

In fact, compare-and-swap is so versatile that it has become the de facto standard building brick of virtually all non-blocking algorithms. Compared to the simple load and store instructions that we have used up to now, it offers an atomic operation that can both retrieve and update shared information. It should not be difficult to see why compare-and-swap is such a powerful operation; it combines three of the most essential actions of the abstract machine: reading, writing and branching.

2.3.1 A counter with compare-and-swap

Implementing a counter with compare-and-swap is straightforward, as can be seen in Figure 2.12. The principle is simple: first, we read a snapshot of the value, then apply any operations locally, and finally update the object in one shot with a single compare-and-swap operation.

What is perhaps more interesting is the lock-free property of the *increment* code presented. Indeed, contrary to previous examples, this method is not wait-free. There is no guarantee that the loop will not go on forever, but with each additional iteration, with each failed compare-and-swap, another thread advances, another thread succeeds in changing the value of the counter. While each winner can easily fall prey to a maverick scheduler just after having altered the shared value, this cannot last, and ultimately, the surviving thread will make progress on its own, thus bounding the number of steps required to push the system forward. This implementation is lock-free.

```
int increment()
{
   bool ok;
   int n;
   do {
      n := counter;
      ok := cas(&counter, n, n + 1);
   } while (not ok);
   return n;
}
```

Figure 2.12: Counter with compare-and-swap

2.4 Introducing the ABA problem: a multi-word counter?

Now that we have built a reasonable single-word counter, we might be tempted to think that extending the feat to a multi-integer counter would be a simple matter. Imagine we want to count over more than the size of a word—let us say a word is 32 bits, maybe we want to be able to count up to 2^{128} instead.

More generally, we often find ourselves in the following scenario: we have an object of fixed size, that corresponds to a known set of bytes, which has been decided upon creation—or any time before any contention occurs. As usual, different configurations of the bits signify different values. Typically, we want to write a method that appears to atomically read the whole multi-word blob, computes a modification, and then updates it. To sum it up, perhaps unsurprisingly, we want most operations on more complex data structures to behave just like some sort of read-modify-write macro-instructions.

2.4.1 Partial read and write operations

The task presents many difficulties centered around the lack of multi-word primitives. Atomic read-modify-write operations only allow us to read one word, which generally does not reflect the overall state of the object. Moreover, since there is no instruction that writes every word of an object at once, there will be times when the shared state represents an operation in progress. What do we do if we read an intermediate state? For starters, how do we even know whether what we have read is not supposed to be a complete value? And finally, how do we manage conflicting updates?

Going back to our multi-word counter example, let us take a moment to look at two non-working solutions. Suppose we allocate all the space to the counting itself in a simple "pure binary" fashion: with the successive bits representing growing powers of two. In that case, inevitably, the less significant words will need to wrap, as the higher octets change, to represent bigger values. The lower sequence 10 appears in the representation of both 2 and 6. We rarely think of it this way, but, after all, the less significant bit flips once for each incrementation.

Since we can only update one word at a time, inevitably, some instruction leaves the counter in an intermediate state. Any thread that observes the counter value at that time sees the transient bit pattern. For example, assuming bit-sized words, is 010 (in binary) the number 2? Or is it a transition between 011 and 100 where the rightmost bit was cleared first? Since every bit combination represents a valid state, it is impossible to tell whether a state corresponds to an update in progress or a legitimate value, if we are only allowed to read from the pure binary representation without any supplementary information. Consequently, it appears we cannot purely rely on such a compact encoding.

The problem described here stems from a need to update multiple words atomically. To get rid of this, we may resort to other representations, such as the reflected binary, or Gray, code. It is true that in a Gray representation, each value differs from the previous one by exactly one bit, which reduces the width of each macro-modification to a single word.

Unfortunately, we still face another obstacle: the need to read multiple words in order to decide in which state we are. Certainly, each transition requires writing exactly one different bit, but we still have to look at the entire sequence to decide which one to update. For example, the lower bit sequence 11 in a three-bit Gray code can belong either to 011 (representing 2) or to 111 (representing 5). Thus, it is still necessary to look at all the bits—hence all the words—to figure out in which state we currently reside, and which one comes next.

Consider the following scenario. Suppose we have a way of incrementing our multi-word Gray register. At some point, the method needs to write the new value by changing the appropriate word in the compound while leaving all other bits unchanged. However, we recall that given our arsenal of load, store and compare-and-swap, anything committed to memory—by way of a store or compare-and-swap instruction—is conditional on at most the value of a single word (in the case of compare-and-swap) or no value at all (in the case of the simple store). Imagine that instead of changing 0001 to 0011, we instead turn 0100 into 0110 (in binary, assuming word size is one bit). The value of the second least significant bit is the same in both states, so one-bit compare-and-swap would know no better. Because of lock freedom, the scheduler might decide to stop the thread right there, thus leaving the counter with an incorrect value that will appear to every other thread as if it were legitimate. Therefore, even in the case of such reflected coding, more space is still required to distinguish between intermediate and complete states.

2.4.2 Compare-and-swap monotony

To give us some insight into a potential solution to the multi-word counter problem, let us consider a non-wrapping—hence finite—counter that saturates at some maximal value. A simple yet effective implementation of such an object uses a unary representation, and thus requires as many bits as the value we want to count to. Its code is given in Figure 2.13. The code is indeed naive. We might point out that since the loop index uses a primitive integer type, our counter is no more powerful—and a lot more wasteful—than the simple one-word counter above. However, perhaps unsurprisingly, the same technique works where the individual word-sized pieces are not booleans but full-fledged integers

```
bool counter[L] := {0};

// Let us not bother with how to read the value, for now. We simply
// return true if the operation succeeded, or false if the counter
// is saturated.
bool increment()
{
    for (var i := 0; i < L; ++i) {
        var b := counter[i];
        if (not b) {
            if (cas(&counter[i], false, true))
                return true;
        }
    }
    return false;
}
```

Figure 2.13: Unary counter

instead, as shown in Figure 2.14. This way, we can represent $(2^w - 1)L + 1$ (where w is the width of a machine word) values with L cells.[7]

The basis for these implementations is a kind of **word-level monotony**:[8] given two counter states represented by their constituent words $c = c_0 \dots c_{L-1}$ and $c' = c'_0 \dots c'_{L-1}$, if $c < c'$ (we reach c' from incrementing c), then $\forall i,\ c_i \leq c'_i$. Intuitively it works by never reusing bit patterns larger than a word in size, which eliminates the need for multi-word—either read or write—operations. This ensures that delayed compare-and-swap instructions are benign: if the general state has moved forward too much, then compare-and-swap simply fails due to its old expected value, which is guaranteed never to occur in more advanced states.

As with previous algorithms, our saturated *increment* method loops until it succeeds in writing a new state computed from what it has read a few lines above. We have extended the read-modify-write principle to multiple words, but lost the ability to wrap around. Indeed state transitions still occur in a single (successful) compare-and-swap statement. Resetting the counter, however, requires all the saturated words to go back to zero for the operation to complete, with all the troubles it entails, as described above.

[7] Incidentally, for unrelated reasons, this kind of memory layout is reminiscent of what is done in variable-length integer codes, such as in Huffman compression, and, to some extent, character encodings such as UTF-8. In those codes, the purpose is to write some preferred values in less space than others. To achieve this, while allowing decoding, no complete sequence may be a prefix of another, longer, string. This leads to schemes where some words (or, more realistically, bit segments) never wrap around, with higher values being preceded by a saturated prefix. In the simplest case, the result is akin to what we have devised.

[8] This is a personal term and not widely used or acknowledged by the community.

```
int counter[L];

bool increment()
{
   for (var i := 0; i < L; ++i) {
      for (;;) {
         var x := counter[i];
         if (x = INT_MAX)
            break;
         if (cas(&counter[i], x, x + 1))
            return true;
      }
   }
   return false;
}
```

Figure 2.14: Word-monotonic counter

There are many ways to get this wrong. For example, if we start by naively zeroing words in ascending order (see Figure 2.15), a competing call might erroneously increment one of the new zeros before the wrap-around is complete, as shown in the trace in Figure 2.16.

2.4.3 Understanding multi-word updates as a permission problem

As a matter of course, when executing any method, a thread makes various assumptions about the object it is manipulating, starting from the preconditions to the call. This is especially important at write statements, as any change performed is carried out on the basis of those assumptions.

For example, in the monotonic counter above, each instance of compare-and-swap relies on the fact that if the comparison succeeds, then not only the target location has the expected value but the whole object is in the corresponding expected state. It is a kind of indirection: we use a single-word value as a proxy for a larger predicate over the entire object.

In the context of compare-and-swap, the mismatch between assumptions made at the time when the expected value is read and at the actual compare-and-swap instruction usually takes the form of the **ABA problem**. In this case, a variable changes from some value A to B, then back to A again, thus appearing identical to single-word compare-and-swap, although the associated state differs, and the initial assumptions do not hold anymore. Again, we recall that, incrementing from 2 to 4 in our pure binary counter example resets the least significant bit to $A = 0$, although, to do so, we must go through 3, in which it is set ($B = 1$). A one-bit compare-and-swap would not be able to distinguish between 2 and 4 based solely on its innate comparison. Most commonly, though, the ABA

```
int counter[L] := {0};

void increment()
{
    for (;;) {
        for (var i := 0; i < L; ++i) {
            for (;;) {
                var x := counter[i];
                if (x = INT_MAX)
                    break;
                if (cas(&counter[i], x, x + 1))
                    return;
            }
        }
        for (var i := 0; i < L; ++i)
            cas(&counter[i], INT_MAX, 0);
    }
}
```

Figure 2.15: Bogus wrapping almost-word-monotonic counter

problem manifests itself in the context of pointers, as we will shortly see in Section 2.5. By extension, we refer to the entire family of assumption-mismatch difficulties as ABA problems, although they may not technically qualify as such (i.e., the target variable itself may be stable, yet assumptions be broken by changes to other parts of the object).

Generally speaking, when writing a new value to a shared variable, we must make sure that the object state thus produced satisfies every predicate currently assumed by other threads. This typically ranges from almost nothing, if a concurrent invocation is just starting, to very precise constraints if it is about to attempt a state-changing modification.

We can see now why the two dense representations we first considered, pure binary and Gray code, are unsatisfactory, in a more precise sense. In both cases, we need to examine every bit in order to compute the next state. Due to the absence of atomic multi-word read instructions, there will always be a gap between the moment we finish reading and have decided on a state—this is our assumption—and the point where any actual modification takes place. Since the representation is dense, every bit already carries an integer-value information, which leaves no room for supplementary signaling. Thus, the assumption cannot be made known to other threads. Therefore, by default, we have no choice but to ensure that any new value we write places the counter in a state which satisfies all possible constraints assumed by other threads. This, in turn, translates to the possibility of another thread requiring any arbitrary object state, including one that is incompatible with our change (i.e., does not contain the value we are writing at the

```
$\langle T_1 \rangle$ increment() {
    $c_0$ := counter[0] ⇒ INT_MAX;
    assert $c_0$ = INT_MAX;
}
$\langle T_0 \rangle$ increment() {
    cas(counter[0], INT_MAX, 0) ⇒ true, INT_MAX;
    cas(counter[1], INT_MAX, 0) ⇒ true, INT_MAX;
}
$\langle T_1 \rangle$ increment() {
    // Now, $T_1$ attempts to load counter[1] and sees 0.
    $c_1$ := counter[1] ⇒ 0;
    assert $c_1 \neq$ INT_MAX;
    cas(&counter[1], 0, 1);
    // The new state of the object is $(0, 1)$, effectively skipping
    // all values between $(0, 0)$ and $(\mathrm{INT\_MAX}, 0)$.
}
```

In traces, instances of method invocations (whether taken as a whole or fragmented) are preceded by their thread name in angle brackets. In addition, we have replaced variables with unique names representing the values read, and, where appropriate, results are indicated by a double arrow.

Figure 2.16: Trace showing ABA with the bogus word-monotonic counter

location we are writing to). Therefore, all modifications are illegal.

A sufficient criterion that avoids this situation is if we can make sure no concurrent method expects the new value we are about to write. We can formulate it as a permission problem. In order to use a given *expected* argument to compare-and-swap, a thread needs to acquire the permission to expect that value–location pair. Conversely, we only allow writing a new value if no other thread concurrently holds such a permission. This can be ensured, for example, by retrieving every available permission tokens on that pair ourselves beforehand. We can easily imagine implementing such a permission scheme with locks; how it can be made to work in a lock-free setting will have to wait until Section 2.6.

For the time being, let us simply observe how this explains the gap in difficulty between a simple word-monotonic counter and its wrapping counterpart. Non-blocking data structures that allow repetitions must deal with stale values that are expected by other threads and thus temporarily unavailable for reuse. These characteristics tend to drive lock-free design toward linked data structures. Indeed, pointers have a built-in notion of indirection, which can be taken advantage of to circumvent such stale values by substituting equivalent chunks of memory with different addresses. We now explain this strategy in greater details.

2.5 Working with pointers

After mostly dealing with dead-ends and non-working ideas in the previous section, we now start to examine actual solutions that build on our previous conclusions: that building a lock-free data structure with repetitions is most easily achieved by chaining blocks of memory through pointers.

In this section, we study two examples of concurrent object design that use pointers. This is both an opportunity to flesh out what linked structures look like, and to reacquaint ourselves with our old foe the ABA problem, in the context of pointers.

A typical linked concurrent object is organized as a set of **blocks** (or **chunks**) that behave word-monotonically, meshed together through pointers. Repetition is then handled at the block level, by swapping saturated chunks for fresh ones.

We postpone the delicate question of how to recycle blocks for future reuse to Section 2.6. In the meantime, the code shown is intentionally incorrect, in both cases, in the presence of ABA. It could be made to work as is if we are in an environment that offers automatic garbage collection—or, alternatively, if we never free or reuse any memory.

We should keep in mind, though, that it is only a temporary writing artifact introduced in an attempt to make the whole endeavor more palpable to readers unfamiliar with the subject matter. For the more impatient, rest assured that this will lead us into the next section, which will finally offer ways to overcome ABA issues completely, by implementing our own lock-free memory management schemes that do not depend on the existence of some magical garbage collector.

```
int *head := calloc(L, sizeof *head);

R update(A args)
{
    for (;;) {
        var p := head;
        var ok, i, old, new, newblk, ret := transition(p, args);
        if (ok) {
            // Monotonic transition.
            ok := cas(p + i, old, new);
        } else {
            // Reset transition.
            ok := cas(&head, p, newblk);
        }
        if (ok)
            return ret;
    }
}
```

Figure 2.17: Generic single-chunk algorithm

2.5.1 Pointer-based multi-word counters

For small objects and those where modifications swipe across the whole data structure, it is usually simplest to keep all the actual data in a single block, as it is very easily manipulated with a compare-and-swap operation on the pointer, as demonstrated in Figure 2.17.[9]

For the multi-word counter, a direct implementation of this plan could look something like the following: a head variable pointing to a single block of L words making up a word-monotonic counter. Each time the chunk saturates, we swap in a freshly allocated one. This is illustrated in Figure 2.18.

There is not too much to say about this piece of code. Informally, the function linearizes whenever it gets away with a compare-and-swap. Delayed threads may safely continue to iterate on expired blocks, even after they have been replaced at the head (materialized by the *counter* variable).

This is because we never bother to free any previous chunk, which obviously leads to memory leaks, unless we can afford a garbage collector. What would happen if instead we

[9] This is similar to the universal construction for small objects by Herlihy [1990], except we allow blocks to be mutable as long as they satisfy the word-monotony criterion developed before. An argument could also be made that it very loosely borrows some basic ideas from the lock-free normal form of Timnat and Petrank [2014], where the task of computing a set of compare-and-swap instances to execute (where the usual data structure logic lies) and actually carrying them out and taking action based on the feedback (which is rather automatic; as is the case here) are separate.

```
int *counter := calloc(L, sizeof *counter);

void increment()
{
    for (;;) {
        var c := counter;
        for (var i := 0; i < L; ++i) {
            for (var i := 0; i < L; ++i) {
                for (;;) {
                    var x := c[i];
                    if (x = INT_MAX)
                        break;
                    if (cas(&c[i], x, x + 1))
                        return;
                }
            }
        }
        // Block is saturated.
        int *b := calloc(L, sizeof *b);
        if (cas(&counter, c, b))
            break;
    }
    // Should we free c?
}
```

Figure 2.18: Single-chunk multi-word counter

freed the swapped-out block at the end of the loop (where indicated by the comment)?

The answer depends on the semantics of freeing. If it invalidates the address and makes it impossible to dereference the associated pointer (e.g., as does the standard *free* call in the C language), then the algorithm becomes incorrect, other threads might still be walking through a previous chunk.

If it does not, and the memory stays accessible transparently by the object methods (e.g., we use a slab allocator where chunks conserve their sizes and types), then we risk running into another problem. Assuming freed pointers become available again for allocation through *calloc*, we are vulnerable to ABA: a delayed thread might swap out an incomplete block at the same address as a previously saturated one.

2.5.2 Other pointer-based counters

While this first pointer-based version of the multi-word counter allows for wrap-around behavior, it still sports some rather bad memory efficiency: with L-sized chunks, we are only able to store $2^w L$ different values. The use of monotonic chained blocks is not tied to this representation, however. Here, we briefly consider some more advanced implementation options. Although none of them is essential to the comprehension of further sections, they may provide some insight into issues of representation and effective memory use in a lock-free environment.

In general, more compact schemes also have shorter monotonic cycles. A direct extension of the previous algorithm is to use exactly the same layout, with a different encoding. With a normal pure binary representation, the need to swap the head pointer is driven by the wrapping around of the least significant word: other cells can only reset in tandem with the lowest one. Conveniently, that is also the only time when multiple words need to be updated at once, which can be hidden behind the pointer swap. That is one swap for every 2^w increments, which might or might not seem much, depending on the application.

Because of the cascading reset behavior of our pure binary representation, there is little need for us to try to give individual words separate chunks—and have multiple head pointers—under that encoding. We might be tempted to mix the two solutions: can we construct a counter with “wide digits” of M words each that can represent up to $2^w M$ values each? Well, certainly, if we so wish (although recovering the actual value becomes more difficult, then): it is like counting in base $2^w M$ at the upper level, and simply a word-monotonic counter at the lower digit level. However, looking at it, it simply makes reset cycles longer; it does not change the cascading behavior in any way, and as such does not warrant a change in the number of head pointers.

2.5.3 A stack of user-provided cons cells

The previous example showcases a single-block object; but what if we want multiple chunks? A prime motivation for such a request is varying-size data structures, of which the list-based stack due to Treiber [1986] is perhaps the simplest, yet most convincing, illustration. Its code is shown in Figure 2.19. At a glance, it appears to use the same familiar principle as before. Each method builds a new list from the value it sees, then attempts to swap it with the previous list. Here, the key is that, although the structure contains many nodes, the state of the object is represented by the single pointer variable, *top*, which locates the head of the list.

However, identifying linked lists by their head pointer is only reliable if the *next* fields of the cons cells do not change arbitrarily when we are not looking. To this end, at the very least, it must be that client code does not touch those memory locations that are internal to the shared data structures. This is assumed as part of non-interference, as described in Section 2.1.1. In general, if we allow mutable links, then a same address may refer to different actual lists, depending on the contents of the nodes.

Of course, we could argue that at any given time, a pointer represents only one list. This is indeed true, at least in the interleaved world of sequential consistency. However, our basic compare-and-swap strategy is not, on the whole, atomic: it relies on iterating around a read-compute-commit loop. In this scheme, the correctness of the computed value in the second step depends on the data loaded in the first. In the terms of Section 2.4.3, this is an assumption we make before going into compare-and-swap, and other threads need to account for this when performing their own modifications.

Even though it might have become rather repetitive by now, let us take a couple of minutes to review why this is an issue in this specific case. The compare-and-swap instruction only branches depending on the value of the pointer, not the pointed-to contents. We do not make a deep copy of the list. Yet, looking at the *pop* method, our computation function—that maps each value to the next—depends on said contents: it reads the *next* field to find the tail of the list. Between the load and compare-and-swap statements, the scheduler is free to inject steps from other threads. A successful compare-and-swap ensures that the address of the cell does not change, but for our algorithm to be correct, there needs to be an additional guarantee that those remote instructions do not alter the value of the *next* field.

As described above, we will always suppose non-interference. We can thus concentrate on our own code for sources of conflicts. The only spot that can defeat a *pop* computation appears to be the assignment to *next* in the *push* method. For trouble to occur, we need the *node* variable in *push* and the *t* variable in *pop* to refer to the same object—otherwise, the two *next* fields would obviously not refer to the same location. One way this could happen is if a previously removed node is inserted into the list again, as shown in Figure 2.20.

In this example, the old node address reappears before the invocation of *pop* in thread T_0 finishes, yet its contents is different; hence, compare-and-swap will succeed even though the assumption that the computation is valid depends on a falsified hypothesis, namely that the node t_0 has not changed. The code is fooled by the presence of a superficially identical value. This phenomenon is perhaps the most common form of the the ABA problem.

As we have seen, bad things happen because we lack word-level monotony on the mutable fields: the contents of both *top* and *next* may cycle—through the reintroduction of previous cells through *push*—before a delayed thread gets a chance to complete its

```
struct Node {
    Node *next;
    int data;
};

Node *top := null;

void push(Node *node)
{
    bool ok;
    do {
        var t := top;
        node.next := t;
        ok := cas(&top, t, node);
    } while (not ok);
}

Node *pop()
{
    bool ok;
    Node *t;
    do {
        t := top;
        if (t = null)
            return null;
        ok := cas(&top, t, t.next);
    } while (not ok);
    return t;
}
```

Figure 2.19: Treiber stack

```
$\langle T_0 \rangle$ pop() {
    $t_0 \coloneqq$ top;
    assert $t_0 \neq$ null;
    $n_0 \coloneqq t_0$.next;
}
$t_0 \coloneqq \langle T_1 \rangle$ pop();
$\langle T_1 \rangle$ push($t_1$);        // Pushes some new node $t_1 \neq n_0$.
$\langle T_1 \rangle$ push($t_0$);        // Reuses $t_0$ and pushes it.
$\langle T_0 \rangle$ pop() {
    // At this point, $t_0$ points to a node with next field equal to $t_1 \neq n_0$.
    cas(&top, $t_0$, $n_0$);
}
```

Figure 2.20: Trace showing ABA with the Treiber stack

operation, therefore getting confused. In the pointer-based counter, this was assumed to be guaranteed by the memory management system—or by never freeing memory. However, the Treiber stack offers a new perspective on the problem: the chunks, here, are not allocated directly by the algorithm, but provided by the user as part of the interface. Consequently, we have no control over block reuse; we have no option to even let memory leak, and even in the presence of a potent garbage collector, the user is under no constraint to free the popped cells. It could definitely be turned into a mandatory prerequisite of our interface: that pointers retrieved be cycled through the allocator (supposing that the garbage collection system tracks all previous uses and does not reallocate said block before it has been cleared of previous references, including delayed ones threatening compare-and-swap). However, we suspect it would appear very counter-intuitive, and awkward to use, to many.

As is often the case, the problem can be worked around by not taking direct chunks from the caller, and instead making internal copies to privately allocated blocks. We then end up in a similar scenario to the pointer-based counter. Although it may not be much to the taste of veteran C programmers accustomed to pointer-passing interfaces, such changes are often a necessity in the world of lock freedom. Indeed, even when applying the various techniques that we are about to see, it is only possible to detect whether a pointer is safe or unsafe to reuse. We can never force another presumably delayed thread to make progress—perhaps it exploded already—nor can we be sure it will not resume normal operation at some point—maybe the operator could salvage it after all. Therefore, a pointer-passing interface would need to return a new kind of error telling the user that this particular choice node is ill-fated and cannot be inserted back into the list, yet. It would then be up to the caller to either wait or allocate a new node to hold the data. While such a contract may be acceptable between components of a close-knit system, it may not be suitable as an outward abstraction.

With this, we have a working multi-word counter, and once again, we postpone the need for solving the much more complex problem of true multi-word updates. Since such multi-word atomic read-modify-write constructs are the most powerful instructions that we know of in the lock-free arsenal—short of full transactions, they could certainly be used to devise solutions to virtually every non-blocking problems, even the most mundane. However, it is our humble opinion that doing can lead to less efficient algorithms, and most importantly, would not help much in understanding some of the more important principles behind lock freedom. Whether we are willing to trade such losses for more automatic methods is naturally up for debate; however, given that this thesis discusses a manually crafted lock-free algorithm for the evaluation of Kahn process networks, we should naturally be somewhat biased toward explaining the little details that make up part of the more traditional process of lock-free algorithm construction.

2.6 Waiting out ABA

In this section, we finally deal with what we consider the main source of low-level difficulty in lock-free programming: dealing with the ABA problem and the complexity it brings to resource management.

As the heading might suggest, to put it bluntly, solving the ABA requires waiting it out. From our initial imperative pseudo-language, we remark two important facts:

1. there is no observational difference between indefinite delay and actual, terminating, scheduling faults: at any point in time, the scheduler might decide to wake up a long asleep thread;

2. there is no way for a thread to directly inject code into or otherwise influence the control flow of another thread.

As we have seen, item number one, in particular, implies that once a thread is about to apply an unconditional (store) or semi-conditional (partial compare-and-swap on one word in a multi-word object) modification to memory, it represents a dormant threat to others that need to be addressed properly. Rule number two further reduces the spectrum of possible actions to basically just one: waiting. But, did we not agree that waiting was against lock freedom, by design?

In truth, it depends on how we wait: what we do in the meantime, and what happens if the awaited time never comes. Busy waiting certainly is against our precepts, because one thread finishing its critical section is essential both to itself and all of its waiters to advance. Generally speaking, we want the kind of wait that satisfies the following criteria:

1. it should allow waiting threads to work on other things while they passively keep an eye out for an opportunity;

2. the locked-up resource must be non-essential to the system as a whole.

The first point simply states that there should be a way to poll without being sucked into a busy waiting loop. The second requirement is more interesting. It has two corollaries.

First, since threads must be able to continue making progress even in the worst-case scenario where the resource ends up unavailable forever, it must be replaceable. In most cases, we will be content with purely **fungible** objects such as memory chunks, i.e., objects that can be substituted for one another based on generic quantifiable characteristics such as size or value. The other kind of design that would be allowed under this rule is one where resources exist in specialized (e.g., thread-specific) copies. However, in such scenarios, waiting is often either optional and included as a matter of performance optimization, or falls back on a pool of entirely fungible objects after exhausting a few fast paths.

Second, assuming a finite system, it implies any **losses must be bounded**, lest we should run out of resources; thus, under normal conditions, unlimited leakage leads to blocking behavior.

2.6.1 Bounded unfairness

Before dealing with the real thing, we first take a look at a more forgiving fault model: **bounded unfairness**. Under bounded unfairness rules, simply, the adversarial scheduler is not free to delay threads forever. There are many ways to describe such a setting; a simple approach is to say that a k-fair sequentially consistent scheduler chooses a same thread T at least every k moves, i.e., that in any history H, there is no k-length segment of H where T does not appear.

If we can guarantee as much—which is normally offered by real-time operating systems, then various strategies are possible. Basically, we need enough time to pass between two uses of a same value, so as to be sure that any stale threatening compare-and-swap instance is gone. But how much time is that exactly? Unsurprisingly, it depends on the k parameter, but also on the algorithm, more precisely on the maximal distance between a load and the corresponding compare-and-swap instruction it feeds into. The safety gap between two uses must be enough for any planned yet already failing compare-and-swap statement to complete and its expected value be reread from shared memory. Therefore, it is not a simple matter of adding idle time before write statements, for such instructions would themselves add to the length of the loop that feeds into compare-and-swap.

Instead, we should refrain from reusing those values that have not done their time. A popular technique consists in **tagging** pointers. In this context, a tag is a modification of the pointer value to make it distinct from previous instances of itself. It usually takes the form of a counter embedded in the few least significant bits of addresses that are aligned to some higher boundary. For example, if we limit ourselves to manipulating objects aligned on four octets, then it leaves the lower two bits unused. See Figure 2.21 for an application of tagging to the Treiber stack. Obviously, this solution is only sufficient if the k parameter is small enough, and compare-and-swap-based loops are tight. This is sometimes assumed to be "good enough" for general-purpose user-space programs—even when no concrete safety bound can be derived—where transient crashes can be recovered in other ways: watchdogs, possibly with automatic snapshots of the application, or higher-level transactions (e.g., in databases).

A different, and perhaps more direct, implementation of the same spacing idea is to

```
// Users must call this function before using a pointer returned from pop.
// Otherwise, pointers should be kept with their tags so as to be passed to inctag.
void *untag(void *p)
{
    uintptr_t addr := (uintptr_t)p;
    return (void *)(addr & ~0x3);
}
```

```
void *inctag(void *p)
{
    uintptr_t addr := (uintptr_t)p;
    addr := (addr & ~0x3) | ((addr + 1) & 0x3);
    return (void *)addr;
}
```

```
void push(Node *node)
{
    bool ok;
    do {
        var t := top;
        node.next := t;
        // Here, we use a different tag to temporize.
        ok := cas(&top, t, inctag(node));
    } while (not ok);
}
```

Figure 2.21: Treiber stack with tagging (changes only)

```
// Per-thread pointer quarantine. Starts with K null pointers,
// where K is the quarantine period.
thread_local NodeQueue *quarant := nodeQueue(K, null);

void increment()
{
    for (;;) {
        var c := counter;
        // Usual word-monotonic increment.
        ...
        // Block is saturated.
        int *b := calloc(L, sizeof *b);
        if (cas(&counter, c, b)) {
            // Put the swapped-out pointer in quarantine.
            enqueueNode(quarant, c);
            break;
        }
    }
    // Free the pointer we swapped out K iterations before.
    // free should be no-op on the initial null pointers.
    Node *old := dequeueNode(quarant);
    free(old);
}
```

Figure 2.22: Pointer-based counter with quarantine (changes only)

explicitly remember and monitor which pointers are susceptible to ABA in a **quarantine.**[10] On the pointer-based counter, for example, it could see each compare-and-swap to the *counter* head variable followed by a registration of the swapped-out pointer in some local data structure for later reuse, as shown in Figure 2.22. Those chunks could be swapped back in after enough instructions have happened to guarantee any delayed thread has completed at least one iteration of the *increment* loop. Our example code is somewhat simplistic in that it counts quarantine days in terms of iterations of the outer loop, which are *de facto* assumed to be equivalent in length, which is not true. In general, though, it should be said that actually counting every instruction is both incredibly tedious (it would most likely need to be done in the form of some automatic instrumentation rather than manual bookkeeping) and ineffective on modern systems and architectures that do not nicely fit into this simple fault model (e.g., because of variable latencies at the hardware level, or unforeseen urgent events at the software level in non-real-time operating systems).

[10] This is a personal term and not widely used; although the concept is widely accepted by the community, to the best of our knowledge, there is no one universally acknowledged terminology for this.

The issue with both the tagging and quarantine approaches lies with their lack of adaptability. The constants that govern waiting times must be chosen in advance and do not change over time. On the one hand, this can lead to an ABA resurgence if the parameters are set too low, even when more resources would be available to circumvent it. On the other hand, if we choose too much safety, we risk wasting those resources even when no ABA threat exists.

Before getting on with more elaborate techniques, let us ponder a moment about what bounded unfairness means for locks. Actually, it should be said that the complaints formulated in the previous paragraph are both alleviated by simply taking a lock for the entire method. After all, if unfairness is bounded, we can simply wait for our turn to perform an operation on the shared object. However, doing so obviously precludes parallel work, though admittedly there should not be much of it to begin with, in the two cases we are contemplating.

Improving on the simplistic quarantine idea, an arguably better strategy is thus to passively wait for other threads to signal when they have made progress, instead of relying solely on the local program counter. Looking again at the code of Figure 2.22, we can see that the time spent by a pointer in quarantine starts when it is swapped out, and ends a number of local steps after that. As we have established before, this should be an upper estimate on the time it takes for every pending *increment* iterations to complete and reload a fresh value of *counter*. So the important event we are waiting for is the completion of concurrent iterations. Then, should we not have threads directly signal whenever they finish one repetition of the loop? This is the basis behind another very popular technique: **epoch-based reclamation**, also known as **read-copy-update** [McKenney and Slingwine, 1998].

Figure 2.23 shows how such a strategy finds its place in the counter code. There are four hooking points for epoch routines:[11]

- at the beginning and end of each iteration, before the counter value is reread;
- after we successfully update the structure;
- and at the end of the method, when we look to move blocks out of the quarantine.

Essentially, an epoch mechanism acts as if we had access to a global clock that synchronized across the entire system, albeit with an interface limited to four predefined functions and with opaque time stamp values that cannot be otherwise manipulated. The *newEpoch* function associates an *Epoch* object with the current point in time—such epochs are totally ordered by the hypothetical global clock. The *beginEpochSection* and *endEpochSection* mark a threat section: at their core, they only acknowledge the current time in the current thread. The former states our intention to read sensitive values from this point onward; the latter indicates that we are done touching blocks for now. We use

[11] We might need more or less depending on the exact flavor of epoch-based reclamation, e.g., one at the beginning and end of the method to signal an active section. For our expository purpose, this simple scheme suffices.

```
// Epoch interface.
void beginEpochSection();
void endEpochSection();
Epoch newEpoch();
bool epochOver(Epoch);

// Quarantine that contains nodes to be freed later.
thread_local NodeMap ∗quarant;

void increment()
{
    for (;;) {
        // Signal that we want to manipulate managed blocks.
        beginEpochSection();
        var c := counter;
        // Usual word-monotonic increment.
        ...
        // Block is saturated.
        int ∗b := calloc(L, sizeof ∗b);
        if (cas(&counter, c, b)) {
            // Register the swapped-out node for future reuse and increment the
            // global epoch, to track threads that might still be holding the c
            // pointer just swapped out. Any thread entering the loop after this
            // point cannot see c at all.
            insertNode(quarant, c, newEpoch());
            endEpochSection();
            break;
        }
        // Signal that we are done with an iteration, and thus hold no
        // ABA-susceptible pointer.
        endEpochSection();
    }
    // Free previously swapped-out pointers if other threads have passed its epoch.
    for (var old, ep : quarant) {
        if (epochOver(ep)) {
            removeNode(quarant, old);
            free(old);
        }
    }
}
```

Figure 2.23: Pointer-based counter with epochs (changes only)

```
typedef unsigned int Epoch;

// The number of calls to newEpoch; may wrap around.
Epoch epoch ≔ 0;

// Indicates which threads are currently in a method covered by our epoch system.
bool contends[THREAD_MAX + 1] ≔ {false};
Epoch records[THREAD_MAX + 1] ≔ {0};
```

For this to work with simple unsigned integers, we assume that threads do not drift apart more than half *UINT_ MAX* epochs. Note that only threads that have their *contends* bit set participate in polls, so others, running outside of any managed method do not need to be accounted for when looking to free blocks.

Figure 2.24: Simple wrapping integer-based epochs: declarations

endEpochSection as a kind of fence, to assert that the current thread will not execute any compare-and-swap or similar operation in the future with values it read before the fence. Finally, the *epochOver* test returns true if every threatening thread has signaled its presence, using *endEpochSection*, at least once in the time before the call but after the reference point passed as argument.

Naturally, we do not want to depend on the existence of an actual global hardware clock. There are many ways to implement an epoch mechanism. In the case of bounded unfairness, we may simply use a wrapping integer, and rely on the guarantee that threads never drift apart more than half the value range, so we can always distinguish between a wrapped time stamp and an actual small one. The shared state for such an implementation is commented in Figure 2.24, and the methods are given in Figure 2.25. Otherwise, we could also simply wait for other threads to catch up whenever we reach the wrapping point, since the wait is known to be bounded. Other approaches exist using system timer interrupts, or alternating bits [Desnoyers et al., 2012].

Generally speaking, read-copy-update and its variants, such as hierchical read-copy-update [McKenney, 2008], have found many uses in modern low-level general-purpose code, notably in operating system kernels such as Linux. Most often, they use techniques based on system timers and interrupts, which are not included in our implementation language, as defined in Section 2.1.1. It should be quite evident that the inclusion of a reliable timer interrupt that strikes periodically in every thread with a known frequency renders all questions of lock freedom trivial, as threads can use it as a barrier to synchronize, effectively waiting for each other in bounded time. An in-depth discussion of epoch-based reclamation would, therefore, stray quite far away from lock-free programming, and is beyond the scope of this introduction.

The three algorithms described above all handle value reuse (or, in the most common case, memory reclamation, in the form of reuse of allocated blocks), although all three

```
void beginEpochSection()
{
    records[threadid] := epoch;
    contends[threadid] := true;
}
```

```
void endEpochSection()
{
    records[threadid] := epoch;
    contends[threadid] := false;
}
```

```
Epoch newEpoch()
{
    var current := faa(&epoch, 1) + 1;
    records[threadid] := current;
    return current;
}
```

```
bool epochOver(Epoch ep)
{
    var lower := records[threadid];
    for (var i := 0; i < THREAD_MAX + 1; ++i) {
        if (contends[i] and lower − records[i] < UINT_MAX / 2)
            lower := records[i];
    }
    return lower − ep < UINT_MAX / 2;
}
```

Due to the assumptions made in Figure 2.24, values can be compared by evaluating their difference relative to half *UINT_MAX*. Since actual epoch values (projected into the natural integers) never differ by half *UINT_MAX* or more, any bigger gap means the subtracted value is higher.

Figure 2.25: Simple wrapping integer-based epochs: implementation

derive their bounded-loss guarantees from the explicit limit on delays between threads. Among those, epoch-based reclamation is especially appealing because of its adaptability: we do not need to tune any delay parameter in advance, objects are freed in a timely fashion, and the system supports as much delay as it has memory.

To go beyond that, and handle cases of unlimited unfairness, while still ensuring finite losses, we need to take a closer look at what makes epoch-based techniques block when scheduling faults are introduced.

2.6.2 Block life cycle

In epoch-based reclamation, each block b cycles through a series of stages:

1. It starts somewhere in the memory allocator.
2. It is then returned to one thread in particular, through *malloc* or a similar call. At this point, b becomes exclusive to its caller thread T.
3. It is swapped into the data structure, making it visible to and open for access from other threads.
4. It is eventually swapped out from the data structure at some time t_b, and enters a quarantine period. After this point, no other thread may read b from the shared object.
5. When it is determined that other threads have relinquished references to any blocks read from the data structure before t_b—and thus to b in particular, then b becomes exclusive to T once again.
6. Afterward, T may free b back to the memory allocator.

It is an exclusive–shared life cycle: a chunk can be reused as a new value when it is guaranteed not to appear in the shared object or locally in other threads. In fact, read-copy-update originated as a shared–exclusive (or reader–writer) locking technique, rather than the specialized resource reclamation scheme we present here. When used in that fashion, an exclusive writer thread installs a new epoch (*newEpoch*) then waits for any pending readers (*beginEpochSection*) to finish (*endEpochSection*).

How does this translate to our setting? The writer part is the easiest: each swapped-out block is waited on by an exclusive thread after calling *newEpoch*. The shared part is less obvious: it is unclear what precise blocks are being used by threads in a given reader section delimited by *beginEpochSection* and *endEpochSection*.

The strangeness of epoch-based reclamation lies in how threads do not actually stop doing usual work to wait for the lock they have demanded—they do not wait in the usual, busy, sense. Instead, every value in quarantine is polled from time to time. The key here is to recognize that the polling reclamation loop actually waits, virtually, not on one read-copy-update lock but on many. Each quarantined block is actually locked individually as part of its life cycle.

The ingenuity behind using this particular locking scheme for reclamation is that it allows lock objects to be mutualized. Instead of signaling for every managed chunk separately, each thread only enters one epoch section that opens the right to read any future values written in the present or in epochs to come, while the section is active.

Whenever a thread declares its intention to read addresses from the shared data structure, by calling *beginEpochSection*, it does not announce which blocks are affected (it actually does not even know), but rather that any value might be held until its read section ends.

There lies the biggest strength and yet also the fundamental flaw of epoch-based reclamation techniques, when it comes to lock freedom. Halting when holding a shared lock on every current and future blocks shared through the data structure inevitably means unbounded memory leakage, which as we have seen cannot be considered non-blocking in a world with limited resources.

2.6.3 Read barriers

In a lock-free world, mutualizing waits on quarantined resources is unwise. We need more fine-grained control over what values are held by each thread at any given time, so that others might flow freely in the meantime.

As we have seen many times now, in non-blocking algorithms, the threat essentially comes from stale values loaded from memory (that could be used later to gate a compare-and-swap statement). Therefore, it stands to reason that loading potentially sensitive values should be protected through the use of additional support instructions. Borrowing the term from the prior works on garbage collection, perhaps we should call this mechanism a **read barrier**.[12] But what do they need to be protected from, or is it others that ought to be protected from them?

A load operation protected by a read barrier, or **safe read** as it was first explicitly named by Valois [1995], at address p should linearizably load a value b at location p and enter a corresponding read section that locks down b for shared access. As usual, we want the two actions to appear as if they executed together atomically. Since for all intents and purposes, we are taking a reader lock on b, it should come as no surprise that once the section has been entered, we may perform the actual load at any time (as b cannot be recycled until we exit the section). But since, contrary to global epoch-based reclamation, each value must be managed separately, how do we know what lock to take before we read the value?

The simple answer is that we do not know, at least, we do not know for sure. As illustrated in Figure 2.26, we can only tentatively enter a read section on some value,

[12] Looking at past work on the related topic of automatic memory management, we can find read barriers in the first incremental (and later concurrent) garbage collectors, such as Baker [1978]'s. In an incremental garbage collector, normal operations (usually collectively referred to as the mutator) are interleaved with steps from collector code that reclaims memory. Without going into details, let us simply say that both actors might alter the shared memory in different ways; in that context, read barriers, that is, code added to protect load instructions, ensure that the system never falls into an irrecoverably inconsistent state. While the particulars are quite different in our case, the fundamental need of handling concurrency between normal accesses and the reclamation procedure persists.

```
T safeRead(T *p)
{
    T x;
    for (;;) {
        x := *p;
        beginReadSection(x);
        var y := *p;
        if (y = x)
            break;
        endReadSection(x);
    }
    return x;
}
```

Figure 2.26: Safe read

load *b*, then see whether they match. If they do, we have got ourselves a complete safe read.

2.6.4 Counting readers in read sections

While read-copy-update provides a convenient mechanism for mutualizing reader–writer locks, having epoch counters for every block in the system could prove quite wasteful.

We first recall that a shared–exclusive lock with writer priority can be implemented using one boolean and one counter, as in Figure 2.27. Briefly, the correction of this reader–writer lock implementation relies on an anti-store-buffering pattern argument, similar to the one we have seen before: if a reader and a writer are concurrent, then either the reader sees the change to the *writer* variable when it checks its value after fetch-and-add, or the writer sees the *nreader* counter go up after it has successfully flipped *writer* with compare-and-swap. Since the design is biased toward exclusive access, once the boolean flag has gone up, pending readers are waited upon but no new ones can queue up, due to the first inner loop in *lockShared*.

Both the *lockShared* and *lockExclusive* methods are clearly blocking, and thus cannot be used directly to build our read sections and reclamation waits. However, the code provides useful insights into how we might implement those.

The *safeRead* function presented above is very similar to the *lockShared* method of our reader–writer lock, with a few differences:

- instead of testing a boolean variable *writer*, the safe read checks the value at location p: the presence of a value x at p indicates that x is available for a read section; instead of waiting on a specific object, a safe read attempts to lock any value read at p;

- the *beginReadSection* and *endReadSection* invocations have been replaced by fetch-and-add instructions that manage a counter tracking the number of readers.

The *lockExclusive* method is also reminiscent of the way reclamation is handled. First, the thread takes ownership of the value through compare-and-swap. Its effect is twofold: it ensures that only one thread gets exclusive access, and, at the same time, warns concurrent and potential future readers that the value is not available anymore for reading. Again, the main difference is what happens when the thread fails to acquire its target. In a normal lock, the thread would wait until the resource is available; in non-blocking resource reclamation, it means instead that the operation has failed (but some other thread has won the right to pursue, as is necessary in lock freedom) and need to be reattempted. Finally, the last loop in *lockExclusive* corresponds to the polling done at the end of methods for the purpose of actually reclaiming chunks that have definitely fallen into our exclusive care.

Based on this comparison, we can deduce one way to implement read sections for lock-free reclamation: through the use of per-resource integer fields that count the number of pending threads that are currently in a read section involving said resource. These can take the form of **reference counters** embedded into the memory chunks themselves, as is done in the works of Valois [1995], and in a slightly different form in Herlihy [1990]'s universal construction using compare-and-swap. Figures 2.28 and 2.29 show the pointer-based counter with reference counting.

We can verify that, with this method, losses are bounded. Since read sections in a given thread do not overlap, even though there are numerous reference counters, each thread only ever holds at most one reference at any one point in time. The quarantines can therefore contain no more elements than there are threads. Indeed, imagine a quarantine in thread T contains two chunks referenced by some other thread T'. Then T' must have ended its first read section (for b_0) at some point before entering the second (for b_1). That point must be after the test of b_0 during the round that swapped out b_1 in T, thus b_1 was already swapped at that point (since the test occurs at the end of the method), making it impossible for T' to read later.

In more complex algorithms, it is possible that threads hold more than one reference at a time, but as long as that number is bounded, and methods regularly sweep exclusive blocks out of their quarantine, this result should stand.

2.6.5 Hazard pointers

The observation that only a small number of sequential read sections overlap is an important one, as it implies most reference counters are zero most of the time. This is wasteful both in terms of memory and synchronization: the memory needed to store the counter, and the powerful compare-and-swap (or fetch-and-add) instructions required to keep them in agreement.

An alternative design to reference counters is to use a dual representation: instead of storing for every resource what threads refer to it, we can store for each thread what

```
bool writer := false;
int nreader := 0;

void lockShared()
{
    for (;;) {
        bool w;
        do
            w := writer;
        while (w);
        faa(&nreader, 1);
        w := writer;
        if (not w)
            break;
        faa(&nreader, −1);
    }
}

void unlockShared()
{
    faa(&nreader, −1);
}

void lockExclusive()
{
    while (not cas(&writer, false, true))
        continue;
    int nr;
    do
        nr := nreader;
    while (nr > 0);
}

void unlockExclusive()
{
    writer := false;
}
```

Figure 2.27: Shared–exclusive lock

```
void beginReadSection(int *b)
{
   faa(b, 1);
}
```

```
void endReadSection(int *b)
{
   faa(b, −1);
}
```

```
bool canBeReclaimed(int *b)
{
   var nref := *b;
   return nref = 0;
}
```

The first integer in the block is the reference counter. The *safeRead* function is defined as above.

Figure 2.28: Reference-counting functions

resources it currently holds. That can be seen as the basis for Michael [2002]'s **safe memory reclamation** technique.

Since this set is bounded (else we would have unlimited leakage if such a thread halts) by what should generally be a small constant, it can be materialized as a per-thread array or list of **hazard pointers**. As shown in Figure 2.30, incrementing a reference counter is replaced with writing the value into a free hazard pointer; decrementing amounts to nullifying the corresponding slot in the array or removing it from the list. Finally, polling for reclamation is done by scanning the hazard pointers of other threads.

In line with our initial motivation, this modification can help reduce both memory consumption (unused reference counters) and store contention (through fetch-and-add). Very importantly, it also moves bookkeeping information out of the blocks themselves, making it possible for the allocator to fuse chunks as necessary instead of operating with constant-size blocks whose meta-data cannot be overwritten.

2.6.6 Optimizations and other reclamation techniques

We now have a strategy for waiting out potential ABA occurrences—which was perhaps the most critical problem to be discussed in this introduction—although several optimizations and alternatives exist. Perhaps most importantly, it should be noted that it is rather unusual for every method invocations to systemically attempt to reclaim resources. It is customary (e.g., in most implementations of read-copy-update, or Michael's safe memory reclamation system) to time the frequency of such collection cycles appropriately so the

```
thread_local NodeSet ∗quarant := ...;

void increment()
{
    for (;;) {
        var c := safeRead(&counter);
        // Usual word-monotonic increment with $c + 1$.
        ...
        // Block is saturated.
        int ∗b := calloc(L + 1, sizeof ∗b);
        if (cas(&counter, c, b)) {
            addNode(quarant, c);
            break;
        }
        endReadSection(c);
    }
    // Free previously swapped-out pointers if their reference counters
    // are zero, meaning there are no more readers and the chunk is not
    // referenced from the shared structure anymore.
    for (var old : quarant) {
        if (canBeReclaimed(old)) {
            removeNode(quarant, old);
            free(old);
        }
    }
}
```

Figure 2.29: Pointer-based counter with safe reference counting (changes only)

```
NodeSet *hazard[THREAD_MAX + 1] := ...;

void beginReadSection(int *b)
{
    addNode(hazard[threadid], b);
}

void endReadSection(int *b)
{
    removeNode(hazard[threadid], b);
}

bool canBeReclaimed(int *b)
{
    for (var i := 0; i < THREAD_MAX + 1; ++i) {
        if (containsNode(hazard[i], b))
            return false;
    }
    return true;
}
```

Figure 2.30: Hazard pointers

cost of ABA management amortizes nicely to a constant, with regard to the algorithm.

As we have discussed before, using larger chunks with simple word-monotonic progressions also increases performance at the cost of more memory and sometimes more complex operations, where elements that would naturally appear as separate nodes in an equivalent sequential data structure are pulled together and handled in a constrained way. This last point, however, also applies to many practical optimizations done on sequential data objects to improve performance-sensitive attributes such as locality.

Other reclamation techniques have also been developped. Two examples are Herlihy et al. [2002]'s **pass-the-bucket**, which shares many similarities with hazard pointers, and Braginsky et al. [2013]'s **drop-the-anchor**, which, in rather simple terms, dynamically associates a single read section with multiple chunks in a connected part of the block graph (which should come with many of the same benefits as using larger blocks).

In conclusion, we can sum up our exploration on the topic of ABA avoidance, as follows:

- The ABA problem can be avoided by waiting for other threads to forget about sensitive values before reusing them. Sensitive values—mostly, chunks of memory—are those that might be used for future compare-and-swap, which would lead to erroneous successes.

- The number of blocks thus reserved should be bounded so as not to block because of a lack of resources needed to finish the computation.

- This wait should be done lazily so as not to block. It is for most intents and purposes a specialized variation of a shared–exclusive lock whose operations may fail instead of spinning in busy loops.

- Loading ABA-sensitive values from shared memory should be protected by a shared (read) section.

- Reclaiming chunks uses the exclusive (write) side of the mechanism.

- Different strategies exist to manage read sections and reclamation waits, including reference counters and hazard pointers.

- Avenues for optimizations include larger chunks, amortized polling, and larger read sections.

2.7 Memory allocation

In this section, we focus on the last piece of the lock-free resource management puzzle: memory allocation.

Let us start with the simplest design possible, that does without any global allocator at all: **per-thread memory pools**. In this approach, blocks are pulled from a thread-local pool and returned to it when they become available for reuse, as indicated by the

ABA avoidance scheme of our choice. This is another example of a very basic partitioning structure, which requires no consensus. Essentially, the pool represents the set of resources ready for shared use (i.e., insertion into the shared data structure), while the quarantine holds those that have not been cleared for reuse yet.

The pool is used to eliminate busy waiting. It is part of the "lazy waiting" strategy we described at length in the previous section: since we cannot stop to actively wait for any single chunk to be cleared, it has to be done incrementally. The pool thus represents the result of that incremental operation. However, it is not the central piece of the system; the prime correctness criterion for ABA prevention is that every cycle of shared reuse is broken by a period of exclusiveness. That way, any value swapped into a shared data structure always appears unique, since it has not been witnessed by any other thread since its last shared appearance.

This fact is essential in understanding how shared memory allocators can be implemented. Indeed, at first glance, we could be tempted to think the role of ABA avoidance mechanisms is to serve as better memory allocators that provide fresh blocks and reclaim old ones to make them new again. In this view, it is difficult to see how the same methods could be used to prevent ABA in the allocators themselves. However, as we have explained, ABA reclamation devices are in fact not directly related to allocators; standalone, they are closer to specialized passive locks: their job is to wait. They ensure that life cycles properly alternate between shared and exclusive.

A shared memory allocator is nothing more than another shared data structure whose purpose is to distribute blocks between threads. It can be as simple as a non-allocating Treiber stack (for constant-size chunks), for example. The stack depends on ABA prevention on the pointers passed to its *push* method: every shared cycle must be broken by an exclusive period; if the same value is passed twice to *push*, between the two invocations, it must not be continuously held by some thread within a stack method.

Initially, the notion of shared and exclusive states is relative to a specific data object. From the viewpoint of the allocator, any chunk that leaves its pointer list and disappears from all of its methods becomes exclusive, although it might very well be shared through some other structures.

A closed fully lock-free system with one main object and one global allocator is essentially a symmetrical construction, with the two exchanging chunks back and forth, passing through quarantines as necessary to wait out any ABA threats.

Managing several separate instances of an ABA prevention mechanism can, however, prove rather wasteful and cumbersome, especially once the system grows to include more objects. Instead, we observe that different instances read sections can be mutualized, as an over-approximation: if a value is exclusive with regard to every possible readers, then it is exclusive with respect to the specific set we need. This is only correct, however, if we can also assimilate writers, i.e., exclusive users. It would be problematic, indeed if multiple threads ran at the same time claiming exclusive ownership. In practice, this constraint is satisfied if blocks never appear in more than one shared data structure at a time—so that only a compare-and-swap instruction at the proper single location is able to grant exclusive access. This might have been a problem were we discussing ABA

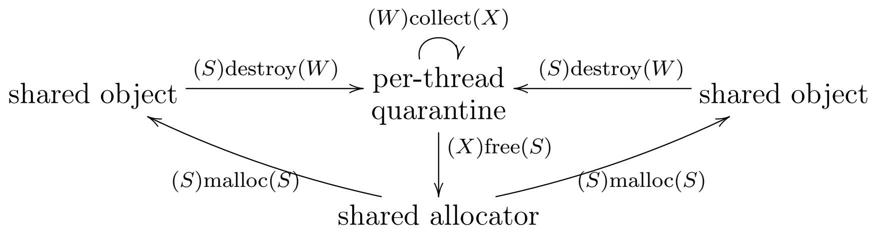


Figure 2.31: System with global allocator and shared objects

prevention for integers, but is barely worth mentioning in the case of pointers, which are naturally used in this way, to indirectly denote unique chunks.

Putting everything together, this all boils down to the convenient fact that we can use a common ABA prevention device to govern memory chunks for use in every shared objects in a system, provided the following rules are respected:

1. at any given time, a chunk is only shared through at most one data structure, including the memory allocator;

2. there is at least one exclusive holding period between any two appearances of a same chunk pointer within a given data structure.

Given that the exclusive periods needed to satisfy rule number 2 are all equivalent, it is usually best to arrange for a centralized quarantine and collection routine at a point in the system that where blocks are guaranteed to pass through in each cycle. A global memory allocator is a perfect place to carry out such a duty. Figure 2.31 illustrates this design, and summarizes the concepts discussed in the previous paragraphs. We have noted the various states that blocks go through during their life, on the diagram, with the letter in parentheses (S for shared, X for exclusive or W when being waited on for exclusive access) before a method name indicating a precondition and after it, a postcondition.

2.8 Disjoint access and the multi-word issue

To wrap up this introduction to lock freedom, we come back to the problem we left toward the beginning of this chapter: multi-word updates. Compared to back then, we are now better equipped with many tools and should already be able to build concurrent data structures to virtually any functional specification we desire, using pointers, chunks, ABA prevention, and memory management. So what is it that we are still lacking? Why multi-word updates?

There is one area where multi-word operations shine: when we want to read and write multiple objects independently. This may seem bizarrely redundant. Of course, multi-word operations handle multiple words! But did we not see earlier in Section 2.5 that the same could be achieved using a few pointers and indirection to large blocks?

The important notion at play is independence. What a linked structure does not offer is efficient **disjoint access**.

Consider the following example. We have two variables x and y, and three concurrent operations: one that changes x, one that changes y, and a last one that swaps the two. Now, under a block-based design, two locations that can be modified by a single method need to share a same block, so they can be updated in one compare-and-swap instruction. Because of the swap operation, x and y thus need to be co-located. However, we would like for the methods that update only x or y to run independently, so that there is no contention between threads that only care about one or the other, unless a swap occurs.

This calls for a multi-word design, where each of the single-variable operations access only that variable, while the two-variable swap spans both, as a multi-word update.

For simplicity, we assume that our data structure is made of protected constant blocks, as explained in Sections 2.5 and 2.6, and that the single-block methods are basic read-modify-write macro-operations as depicted in Figure 2.17.

2.8.1 Descriptors and assistance

As we have touched upon in Section 2.4.1, a central issue when dealing with operations than affect multiple variables is leaving shared variables in an inconsistent, intermediate state, where only some of them have been properly updated. What happens if a thread halts after overwriting x with the value of y without completing the opposite leg of the swap?

From our analysis in Section 2.4.3, we know such transient states need to be made explicit—as additional bits of information in shared memory—so that other threads may learn about them. Since our data structure is made of protected chunks, transient states can be added as a new kind of blocks, different from the ones we use to represent actual values. These intermediate blocks are known as **descriptors**, owing to the fact that they describe ongoing operations.

At the very least, a descriptor must carry enough information to compute one or more desired outcomes, in terms of the shared variables making up the object. For example, the descriptor for our swap operation may be a two-item array, which hold the values of x and y assumed by the original swapping thread at the point where it wishes to write those values back to memory (in reverse positions).

As a thread loading the state of the object, whenever we encounter a descriptor instead of a regular value, we should remember that lock freedom forbids us from waiting for the transient state to fade. After all, this might be the last surviving thread. Therefore, we have no choice but to help move the object toward one of the prescribed outcomes, an act known as **assistance**.

2.8.2 Snapshots

Before we can make and publish assumptions about the state of some object, we first need to load it. When such state consists of multiple words, this is an issue in itself, as we have seen in Section 2.4.1. The general task is known as the **snapshot** problem: we

```
 1  (‘T *, ‘U *) snapshotTwo(‘T **pp, ‘U **qq)
 2  {
 3     var p ≔ safeRead(pp);
 4     var q ≔ safeRead(qq);
 5     if (p = *pp)
 6        return p, q;
 7     else {
 8        endReadSection(p);
 9        endReadSection(q);
10        return null, null;
11     }
12  }
```

We introduce the ‘*T* type variable notation for polymorphic functions. We only allow those types to appear as base types for pointer types. This is just syntactic sugar for *void* pointers, similar to how erasure works for generics in the Java language.

Figure 2.32: Snapshot of two protected pointers

want to retrieve a consistent view of all the variables making up the object. By that, we mean a set of values that have coexisted at some point in time in the current execution.

Correct and efficient snapshot algorithms depend heavily on the underlying data structure and its invariants. For the our toy two-pointer problem, there is a relatively simple solution that relies on read sections, as demonstrated in Figure 2.32. Notice that the *snapshotTwo* procedure also returns read permissions along with the results, if any. In a sense, it works similarly to a safe read. We first read *pp*, then *qq*, then *pp* again. If both values of *pp* match, then we know that there is no modification of *pp* between line 3 and line 5. Indeed, *p* is protected by the safe read, such that for it to occur again in the data structure, it would first need to be removed, quarantined, and freed. Such a sequence cannot happen between the two reads, as we hold a part of the permissions needed to do so, in the form of a read section on *p*. Therefore, on line 4, *pp* contains *p* and *qq* still contains *q*.

2.8.3 Double compare-and-swap

Once we have a consistent snapshot of our object state and a descriptor recording that information, it is time to actually update the variables, in what amounts to a double compare-and-swap, which acts over two possibly non-contiguous words at a same time. The method we sketch here is basically a simplified version of the multi-word compare-and-swap algorithm of Harris et al. [2002], which is, to our knowledge, the most complete modern implementation thereof, and is both lock-free and parallel on disjoint accesses.

Since we have two locations to modify, there is always a risk that one of them changes before we can publish our descriptor; that would mean failure and another attempt.

As a consequence, while writing definitive values to x and y, we need to make sure that both of them do not move from their expected values for the whole duration of the swap. It would be catastrophic to write a new value to x, only to realize we cannot update y because it differs from the recorded snapshot. At that point, a thread working only with x might have already assumed the new value was permanent and continued with further mutations.

In other words, the swap must be ordered with respect to all operations on x and all operations on y. Since those boil down to a single central read-modify-write instruction on either variable, the only way to interpose ourselves in their chains of updates is to overwrite that variable ourselves, with our descriptor pointer.

Given a descriptor d, which describes a snapshot acquired as explained above, the double compare-and-swap is carried out in three stages, as shown in Figure 2.33. We initiate a swap by calling *performSwapXY*, and any reader that stumbles upon a descriptor must invoke *assistSwapXY*.

The *performSwapXY* routine starts by overwriting x with d, at which point the descriptor becomes visible to other threads operating on x, which are recruited to help. The assistance procedure, *assistSwapXY*, consists of two parts: publishing the descriptor to the y crowd and wrapping up the transaction.

One way to explain why this works is to start with the wrap-up. There are two exclusive branches; since *status* moves monotonically from *UNDECIDED* to either *OK* or *FAILED*, if a thread takes the first branch, it is impossible that another enters the second in the same execution.

- If *status* is *OK*, then we require both x and y to have changed from their old expected values to d, at some point prior.
- If *status* is *FAILED*, then we simply cancel out the effect of any previous swap.

As usual, we assume non-interference, meaning that only these two functions move x and y to and from d. In particular, this guarantees that x only changes to d if its previous value matches the associated snapshot.

Being in *assistSwapXY* implies that x has changed to d previously; otherwise, there would be no shared descriptor at all. Furthermore, if a thread sets *status* to *OK*, it must first see y equal to d, which may only happen as a result of compare-and-swap from the old expected value of y, since no other statement overwrites y with d. Therefore, the above *OK* branch is correct.

The *FAILED* branch is always safe, as it is exclusive with the *OK* path. Therefore, any instance of compare-and-swap it performs simply reverts the value of x or y back to what it was when first replaced with d.

2.8.4 Revisiting the in-place multi-word counter

We now revisit and finally bring a solution to the in-place multi-word counter problem.

As we have seen many times now, indirection is king in the world of lock freedom, because of the need to work around quarantined values that, from the perspective of a

```
enum {
    OK,
    FAILED,
    UNDECIDED
};

bool performSwapXY(Desc *d)
{
    d.status := UNDECIDED;
    if (not cas(&x, d.x, d))
        return false;
    return assistSwapXY(d);
}

bool assistSwapXY(Desc *d)
{
    if (d.status = UNDECIDED) {
        if (cas(&y, d.y, d) or y = d)
            cas(&d.status, UNDECIDED, OK);
        else
            cas(&d.status, UNDECIDED, FAILED);
    }
    if (d.status = OK) {
        cas(&x, d, d.y);
        cas(&y, d, d.x);
        return true;
    } else {
        cas(&x, d, d.x);
        cas(&y, d, d.y);
        return false;
    }
}
```

Memory management is omitted to keep the code simple and focused on the main algorithmic elements. The value of descriptor d is supposed unique.

Figure 2.33: Double compare-and-swap for the two-pointer object

given thread, should be reusable, but truly are not due to threats (assumptions) by other threads. In-place operations are no exceptions.

Starting from the word-monotonic counter, the main challenge in the construction of a multi-word counter lies in the need to cycle individual words back to zero once we have reached the maximal value. These are the two dangerous values.

- Since delayed threads might be threatening to increment a zero value to one, assuming a different overall value, it may not be safe to reuse zero.
- Similarly, if we interrupt a thread while it is resetting some words, it might awaken to continue later, when the overall value is not saturated anymore, but individual words have maximal value.

We have now seen several block-based data structures, using pointer indirections; in such systems, there is a notion of equivalence between memory states: only the contents of chunks matter, rather than the absolute pointer values. This concept can be applied to the present case as well. There is a need to distinguish between concrete values with meaning zero, or maximum, in case one of them is held by a delayed thread, yet they should all be equivalent, for the purpose of counting: the successor to any zero is always one, and maximum is always followed by zero.

We thus divide the set of word values in two classes: univocal values, and equivocal ones. Although we could imagine more compact partitioning, for simplicity, we settle for reserving the lowest two bits: 11 (in binary) for univocal and any other pattern for equivocal (as we will see, we need a few different kinds). The first sort is the usual integers, taken at face value when shifted by one bit to the right to eliminate the type bits. The second is the class of all those equivalent values whose meaning is actually zero (type bits equal 00) or maximum (type bits equal 10). Remember that in a tagging scheme (which is not lock-free), we would simply iterate through bit patterns associated with, e.g., zero, by incrementing until we loop, hoping that any pending thread has relinquished any old values by then.

In a proper ABA-protected system, however, delayed thread hold equivocal values through the use of read sections. As we have seen before, the need for different values as well as the potential losses are bounded and proportional to the number of threads; with a sufficiently large value range reserved for the equivocal class, we can be sure that the algorithm is non-blocking, including in terms of resource availability.

The proposed plan solves the problem of pending compare-and-swap statements, but there remains another equally important issue: how do we ensure that lower words are still saturated by the time we attempt to increment a higher word? Similarly, how do we know that we are replacing all the maxima of a single saturation? What if we were delayed and touch maximized words in a non-maximized overall state?

The basic insight necessary to answer this question is that once a read section is entered for a given value, it blocks that value from reappearing in the same place indefinitely until the section is released. Therefore, we can take advantage of the fixed update order of our counter to good effect, in order to take a snapshot of the shared state, as illustrated in the *snapshot* function of Figure 2.35.

- Suppose that resetting to zero always occurs from the lowest to the highest word. We walk through the data structure, taking read sections on each of the maximum values at all positions. Once we reach saturation, we confirm our findings by rereading the first word: if its value is unchanged, then we know that we have got ourselves a proper snapshot of the object. Indeed, if any maximum value loaded had changed since our initial read of the first word, then since it is maximal, it would have required all the lower words to also be maximized (during saturation) or the first word to be zero (during another reset). In both cases, the first word would have been overwritten, which did not happen, since we reread the same value at the lowest cell and are guaranteed by the read section that values do not repeat.

- If instead we reach a non-maximized cell, then either it is non-zero, in which case, an increment is always correct, as any different overall state with the same word value would be equivalent (i.e., lower words maximized, higher words zero). If it is zero, the same trick as above can be used: we read the first cell again. If it matches, then similarly, this zero cannot have cycled since we entered the first read section and loaded the initial value.

With these two patches in place, only the problem of assistance remains. A resetting thread might be delayed during its operation, leaving the half of the words at maximum and the other one at zero. To correct this flaw, we need other threads to step in and complete the zeroing in its stead, should the help be needed.

Essentially, when a thread sees zero in the first word, it looks ahead to see if any maxima remain to be reset, and if so, it offers to help. Again, this relies on the snapshot being accurate, but fortunately, an argument similar to the above applies: if word i is maximum and the first word is zero twice in a row, then i cannot have changed between the two reads of the first word. If it had, then since it is maximal, it would have required all the lower words to also be maximized (during saturation) or the first word to be a different zero (during another reset), which is not the case since the first word is the same zero and cannot have been repeated due to the read section.

Putting all the pieces together, we get something along the lines of Figure 2.34, with auxiliary functions defined in Figure 2.35. Looking at the code, in a way, we have implemented the exact natural semantics of how we first envisioned wrapping should be achieved: by resetting words one after the other when saturation is reached. What we did not foresee was that we now have equivalence classes instead of some sensitive values.

Generally speaking, this new counter uses a slew of tricks and techniques far more advanced than the only unprotected compare-and-swap instruction that we then had at our disposal. Whether it is in any way useful, compared to its pointer-based alter ego is doubtful. However, it shows that all of the concepts we have seen in this chapter can blend together to create complex objects.

2.8.5 Our complete non-blocking toolkit

Snapshots, assistance, and multi-word updates thus complete our lock-free programming toolkit, as this chapter draws to a close. Over the course of this introduction to lock

```
void increment()
{
    int x[L];
    int i;
    for (;;) {
        snapshot(x);
        // If first word is zero, someone might need assistance.
        if ((x[0] & 0x3) = 0x0) {
            for (i := 1; i < L and (x[i] & 0x3) = 0x0; ++i)
                continue;
            if (i < L)
                reset(x, i);
        }
        // Skip maxima.
        for (i := 0; i < L and (x[i] & 0x3) = 0x1; ++i)
            continue;
        if (i = L) {
            // Reset and try again to get the first increment.
            reset(x, 0);
        } else {
            // Attempt single increment.
            switch (x[i] & 0x3) {
            case 0x0:       // zero
                if (casnew(&counter[i], x[i], 0x6)) {
                    cleanup(x);
                    return;
                }
                break;
            case 0x3:       // univocal
                int p := 0;
                if (x[i] = (INT_MAX & ~0x1))
                    p := alloc(0x1);
                if (casnew(&counter[i], x[i], p ? p : x[i] + 0x4)) {
                    cleanup(x);
                    return;
                }
            }
        }
        cleanup(x);
    }
}
```

Figure 2.34: Wrapping almost-word-monotonic counter (main method)

```
int counter[L] ≔ {0};

void snapshot(int x[])
{
    for (;;) {
        for (var i ≔ 0; i < L; ++i)
            x[i] ≔ safeRead(&counter[i]);
        var again ≔ counter[0];
        if (again = x[0])
            break;
        cleanup(x);
    }
}

void reset(int x[], int i0)
{
    for (var i ≔ i0; i < L; ++i)
        casnew(&counter[i], x[i], alloc(0x0));
}

bool casnew(int ∗p, int expected, int new)
{
    if (cas(p, expected, new)) {
        if ((expected & 0x3) ≠ 0x3)
            free(expected);
    } else {
        if ((new & 0x3) ≠ 0x3)
            free(new);
    }
    return ok;
}

void cleanup(int x[])
{
    for (var i ≔ 0; i < L; ++i)
        endReadSection(x[i]);
}
```

Figure 2.35: Wrapping almost-word-monotonic counter (auxiliary functions)

freedom, we have seen several important techniques and mechanisms that constitute the foundations on which our Kahn process implementation is built:

- Explicit lock-free memory management, as described in Sections 2.6 and 2.7. Where some algorithms leave out the memory management details—assuming they can be worked out later using some implementation of read sections—we think it is important to build the system from the ground up with these considerations in mind, as they affect performance and often dictate what kind of data structure layout may be preferable.

- A construction using linked monotonic blocks (Sections 2.3 and 2.5) to produce various compound data objects that do not need disjoint access capabilities. This is very effective for any data structure in which state changes sequentially, although operations may come from different threads. There are numerous instances of these in a Kahn process network interpreter, due to the nature of communication in such programs, as we will see in the next chapter.

- In particular, it should be stressed that word-monotonic updates (Section 2.4.2) constitute a highly efficient concurrent transport, since they do not require any expensive synchronization primitive (even more so in a relaxed setting, as we will see in Section 4.2). Although limited to a finite number of modifications when used by themselves, we have seen how they combine with a block replacement strategy (Section 2.6) to form versatile reusable objects.

- Lastly, the more expensive multi-words ingredients presented in this chapter, such as descriptors, snapshots and assistance, can be used to implement operations spanning multiple monolithic objects while offering disjoint-access parallelism.

As we slowly head into the next chapter, which deals with a fairly large system made of several components, our focus will shift from individual techniques to their applications and combinations: how to design actual lock-free data structures to support Kahn process networks, and how to compose non-blocking objects in a reasonable way.

Our solutions involve hybrid approaches combining linked monotonic blocks, descriptors, specialized multi-word operations, and a fair dose of careful planning around the order and nature of updates, so as to allow assistance. All of our algorithms build on top of the techniques we have presented in detail in this introduction, with a few ideas unique to the making of Kahn process networks sprinkled across, and an overall attention to performance characteristics such as overhead and contention.

Chapter 3

A lock-free Kahn process network implementation

We now present our first main contribution: a non-blocking Kahn process network implementation that is is parallel when working on disjoint parts of the process graph.

As we did previously, the code presented throughout this chapter is written in a syntactic variant of the C programming language, with a few extensions, and assuming sequentially consistent concurrent semantics. We insist that, although the syntax may appear somewhat different from what can be found in usual C source files, the changes are purely cosmetic. Unless explicitly stated otherwise, the actual code shown (aside from the extensions) is intended to be valid C, with its normal precise semantics as defined by the C standard.

3.1 Overview of Kahn process networks

For our purpose, we define Kahn process networks to be the programs that can be expressed in a variation of the concurrent imperative language first introduced by Kahn [1974], and further specified by Kahn and MacQueen [1977]. The syntax is shown in Figure 3.1.

Informally, the language semantics follows Kahn's description, summarized as follows. A program is a network made of a number of **processes**, declared with the *process* keyword (*proc* rule). Those processes do not exist until called by an *inst* rule, which creates an **instance** of the process and binds all its formal parameters to actual channels and concrete values—as would a normal function call in a traditional programming language.

The main constraint that applies to a Kahn process network rules that channels must have exactly one producer and one consumer. It is an error for a channel identifier to appear twice as an in argument, or twice as an out argument in *inst*. Additionally, channel variables that are declared as in or out parameters in the enclosing context can only be bound on that side. For example, if a process reconfigures into two subprocesses, it cannot bind any of its in parameters as an output channel of its children.

```
prog →  proc∗

proc →  "process" NAME "(" procParmList? ")" blck
procParmList → procParm ("," procParm)∗
procParm → (chanType | type) NAME

blck →  "{" decl∗ stmt∗ "}"
stmt →  NAME ":=" expr ";"
     |  "if " "(" expr ")" blck
     |  "while" "(" expr ")" blck
     |  chanStmt
     |  rcfg
chanStmt → NAME "←" expr ";"
rcfg →  "reconfigure" "{" chanDecl∗ decl∗ inst∗ "}"
chanDecl →  "channel" NAME "[" expr "]" ";"
decl →  type NAME ";"
inst →  NAME "(" exprList? ")" ";"

exprList →  expr ("," expr)∗
expr →  prod ("+" | "−") prod
prod →  prim ("∗" | "/") prim
prim →  NAME | "(" expr ")"
     |  NAME "(" exprList? ")"

chanType → "in" |  "out"
type →  "int"
```

Figure 3.1: Kahn process language

It is assumed that some operator in the external run-time environment calls a designated main process at startup, and provides it with input data in the form of constants and data on its input channels, while consuming output from its output channels.

The code within process bodies executes sequentially. It may use local integer variables, as well as **channels** declared with the *channel* (*chan-decl* rule), or received as argument specified with the *in* or *out* keywords (*proc-parm* rule). In addition to the usual imperative statements, the code may perform three concurrent actions:

- reading from a channel, if a channel identifier appears on the right-hand side of the arrow operator (*chan-stmt* rule);
- writing to a channel, if the channel identifier appears on the left-hand side (*chan-stmt* rule);
- and replacing a process with a subgraph, through the use of a *reconfigure* statement (*rcfg* rule).

Channels are the only means of communication between live processes, and reconfiguration is the only way to add new processes to the network.

In a Kahn process network, each process is always either executing a statement or waiting on a channel. Since we implement bounded queues, processes can wait on either a read or a write statement.

3.1.1 Non-blocking Kahn process networks?

We recall from the last chapter that non-blocking systems are defined as concurrent systems that tolerate arbitrary scheduler delays while making progress as a whole. In particular, in a finite system, this also implies bounded resource (e.g., memory) consumption. As we have seen, two important parameters are at play in a lock-freedom claim: scope and progress.

As far as Kahn process networks are concerned, the question tends to naturally raise some suspicion. After all, individual Kahn processes in a Kahn network follow a blocking semantics (they wait for data on channels, and we know how waits are bad for our lock-free karma). Consequently, the notion of a non-blocking implementation might at first appear counterintuitive. How can execution not block if one Kahn process completely stops? The calculation it is responsible for will no doubt cease to be performed.

The key observation here is that **Kahn processes are not threads**. In fact, the situation is very similar to one we have already encountered: the reader–writer lock-like devices we used for resource reclamation in the previous chapter. The same principles of non-blocking waits apply here: we are allowed to wait for some condition, as long as we do not actively do so (busy loops), and the sum of losses due to conditions never materializing (as a consequence of scheduling delays) must be recoverable or bounded.

These requirements already provide two fundamental guidelines as to the design of a lock-free Kahn process implementation:

Multiplexing All threads should be able to run multiple Kahn processes alternatively. Failure to read from or write to a channel should not result in active waiting, but execution should instead pick up another process.

Concurrency Conversely, all processes should be able to run on any thread, at any time, including concurrently. Although at a high semantic level, Kahn process networks rely purely on single-producer single-consumer communication, as far as our implementation is concerned, there may be multiple threads attempting to touch a same process or modify a same channel simultaneously.

There lie the apparent contradictions of our lock-free Kahn process network implementation. We want to write what amounts to a non-blocking interpreter with inter-producer and inter-consumer concurrency, for a blocking single-producer single-consumer process language.

3.2 A sequential interpreter

3.2.1 Single-process interface

We start by looking at a sequential interpreter for a single Kahn process—not a whole network, just one process. The interpreter takes as input a description of the sequential instructions of the process, written in (actually translated from, as we will see) the Kahn process language, and executes it on a target machine, in our case, one of the threads of our concurrent model defined in the previous chapter.

This could be as simple as a single procedure. However, the process can have unbounded input and output channels that communicate with the environment. Since infinite streams of values can be read and written on those transports, they cannot simply be implemented as function arguments or return values.

Instead, it is necessary to design an interface which interleaves Kahn process work and code for input–output from the environment. We postpone a precise definition of what constitutes a proper Kahn process in the context of our interpreter to Section 3.2.2. For the time being, let us simply say that a Kahn process acts as a kind of "coroutine." Intuitively, a coroutine is a routine that keeps its own state and can be interrupted and resumed, alongside its calling context. Notice that this calling context can be another Kahn process, such as the parent process in the case of a recursive process network, or the environment of the interpreter itself, if the process is the top-level root process.

Such coroutines are **cooperative** in the sense that control flows from one coroutine to its caller at certain well-defined points (i.e., channel input and output statements in a Kahn process), instead allowing preemption at arbitrary positions in the code.

Thus, our interpreter should be envisioned not merely as a function, but an object, which maintains a state and has methods to communicate with it, as shown in Figure 3.2. Our interface may deserve some explanation. First, we assume, for the time being, the existence of values of type *Proc*, which represent programs to be interpreted, in a suitable

```
(bool, Chan *) cocall(Proc *);
(bool, int) chrecv(Chan *);
bool chsend(Chan *, int);
```

Figure 3.2: Sequential Kahn process interface

format. Such a process can be evaluated through *cocall*. The *cocall* method executes the coroutine until one of the following observable events occurs:

Termination If the process exits, *cocall* returns true and a null channel pointer.

Input or output When the coroutine performs a successful input or output operation on a channel, *cocall* returns true and the channel in question. This gives an opportunity for the environment to react, if it so wishes.

Wait If instead, the input or output operation fails due to lack of data on read, or space on write, then *cocall* returns false along with the guilty channel.

The *chrecv* and *chsend* functions can then be used to remove data from output buffers (thus freeing space), and provide new data to the process, respectively. In a sequential context, they do not fail unless the channels are empty or full.

Alternatively, the interpreter could also be conceived as a procedure that makes extensive use of callbacks. This is just a matter of flow inversion. We consciously choose to approach it as a stateful object, however, as it will make the transition to a lock-free linearizable data structure that much easier in the next section.

In the case where there are only one input and one output channels, our Kahn process behaves much as would a filter of sorts. Similar to how a Unix pipe works, the single-input single-output Kahn filter receives a stream of data, processes it and spits out a new sequence of values. The only difference—though admittedly a major one—is the presence of a preemptive process scheduler, as well as forced memory isolation, in the case of Unix. With Kahn processes, a single thread of execution has to juggle between the coroutine and its caller explicitly. It is also the case that it must follow the cooperation protocol: we saw in the last chapter that the thread scheduler is adversarial, in an environment that requires lock freedom; in contrast, we consider the local coroutine scheduler in each thread to be under our total control (or at least acting in a friendly way) and perfectly willing to carry out the defined protocol, if the platform or hardware lets it do so.

This serves as the basic abstraction for our Kahn processes. As a data object, the interpreted Kahn program appears as a cooperatively concurrent "multi-filter," whose observable behavior is described by three phenomena:

- data exchanges from the environment to the process, initiated through *chsend* and acknowledged by a true return from *cocall*;
- data exchanges from the process to the environment, initiated through *chrecv* and acknowledged by a true return from *cocall*;

- and termination of the process, as attested by a null return from *cocall*.

3.2.2 Processes as state machines

Although we present our work as the construction of an interpreter, we are not interested in the sequential parts of the evaluation. In fact, we want to allow sequential code written in the target language to be turned into and run as Kahn processes. Instead of a full-fledged interpreter and source language, we provide a **run-time library**. This library accepts programs written in a version of the Kahn process language encoded in the host language as data and functions. We now describe this embedding into a fairly average imperative language with threads, as we have relied on in the previous chapter.

The main intent of such a program representation is, of course, that most statements simply map one-for-one to their host counterparts, in the sequential portions. We assume the target language does not have built-in coroutines. Instead, embedded Kahn processes are written as **state machines**.

- The **state** of a machine is given by (the value of) a dictionary of all local variables declared in the Kahn process, which is most commonly implemented as an array (with variables properly renamed) or structure with named fields. In the simplified version of the language we are studying, those fields are limited to integers, but they could easily be extended to hold linked data structures, making for state machines that are not necessarily finite.

- A **transition** corresponds to a potentially yielding statement, where control might deviate from the normal sequential flow and be transferred deliberately to the caller of the coroutine. These special cooperation points include channel input and output instructions, as well as the special handling of the *reconfigure* statement that will be explained in the next subsection.

Given the above definitions, we can already construct a simple run-time library for single Kahn processes without reconfiguration, which are just coroutines with bounded first-in first-out queues. To get an idea of the whole picture, we give pseudo-code for this first step in Figures 3.3 to 3.7, using the code for the simple queue seen in Section 2.1 (not reproduced).

In our sequential implementation, the state machine is implemented by a combination of a state array and a *step* transition function. The *state* field in the *Proc* structure (Figure 3.3) points to two reserved cells (see below), followed by one cell per process-local variable, as described above. The transition function is expected to comply with certain rules:

- It does not access any shared memory aside from the *Proc* structure it receives as argument. Within that object, only the *state* array can be written; the other fields are read-only.

```
struct Proc {
    bool step(Arg[*], int[*]);
    Arg args[*];
    int state[*];
    bool finished;
};

struct Chan {
    int front;
    int back;
    int buf[*];
};
```

We use the $T[*]$ type to denote a fat array pointer, which holds both the length and the address of the array. It may be cast to and from an integer–pointer pair of the appropriate type. The *countof* operator provides access to the length of a fat pointer. Otherwise, it automatically decays to a pointer of the right type, and behaves just like a normal array pointer over items of type T.

Figure 3.3: Sequential single-process implementation: *Proc* and *Chan*

```
enum ArgKind {
    IN,
    OUT,
    CST
};

struct Arg {
    ArgKind kind;
    union {
        Chan *ch;
        int c;
    };
};
```

Figure 3.4: Sequential single-process implementation: *Arg* and *ArgKind*

```
Proc ∗mkproc(Proc ∗p, bool step(Arg[∗], int[∗]), int nvar, Arg args...)
{
    p.finished := false;
    p.step := step;
    p.args := calloc(countof args, sizeof ∗args);
    memcpy(p.args, args, countof args ∗ sizeof ∗args);
    p.state := aalloc(2 + nvar, sizeof ∗p.state);
    p.state[0] := p.state[1] := −1;
    return p;
}
```

```
void cofree(Proc ∗p)
{
    free(p.args);
    free(p.state);
}
```

The *aalloc* function is a wrapper around *calloc* that returns a fat array pointer.

Figure 3.5: Sequential single-process implementation: *mkproc* and *cofree*

- It does not perform any input–output action, or otherwise communicate with external sources. Thus, it behaves as a pure function over states: it maps one state to the next one, although it does so by mutating the state array.

- It returns in a bounded number of instructions, i.e., transitions take a finite amount of time.

- If the process terminates, it returns true and does not alter the state.

- If it wants to read from or write to one of its process arguments, it returns false, and sets the two reserved cells of the *state* array as follows. The first cell should contain the index of the argument to read or write. The second one specifies the index of the process-local variable to copy date to or from. This initiates a state transition, such that the next time *step* is called, the requested action is guaranteed to have been completed, and its state changed, if necessary, to reflect it.

In effect, the first two cells of the *state* array encode the part of the transition that may require the process to wait, and delegates the actual waiting to the run-time library (see *cocall* in Figure 3.6) instead of entering a busy loop. This allows lazy waiting similar to the quarantine for recycling pointers, in the last chapter. The reserved cells are initialized to invalid indices to indicate that no transition is ongoing, and are guaranteed to reset each time *step* is called.

```
(bool, Chan ∗) cocall(Proc ∗p)
{
    if (p.finished)
        return true, null;
    if (p.state[0] = −1 or p.state[1] = −1) {
        if (p.step(p.args, p.state)) {
            p.finished ≔ true;
            return true, null;
        }
    }
    assert(p.state[0] ≥ 0 and p.state[0] < countof p.args);
    assert(p.state[1] ≥ 2 and p.state[1] < countof p.state);
    var arg ≔ p.args[p.state[0]];
    var px ≔ &p.state[p.state[1]];
    bool ok;
    int x;
    switch (arg.kind) {
    case IN:
        ok, x ≔ chrecv(arg.ch);
        ∗px ≔ x;
        break;
    case OUT:
        ok ≔ chsend(arg.ch, ∗px);
        break;
    default:
        abort();
    }
    if (ok) {
        p.state[0] ≔ −1;
        p.state[1] ≔ −1;
    }
    return ok, arg.ch;
}
```

Figure 3.6: Sequential single-process implementation: *cocall*

```
Chan mkchan(Chan ∗ch, int len)
{
    ch.front := 0;
    ch.back := 0;
    ch.buf := aalloc(len, sizeof ∗ch.buf);
    return ch;
}
```

```
void chfree(Chan ∗ch)
{
    free(ch.buf);
}
```

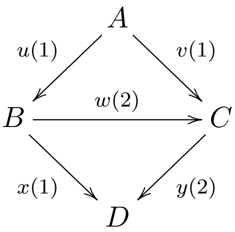
```
(bool, int) chrecv(Chan ∗ch)
{
    return pop(ch);
}
```

```
bool chsend(Chan ∗ch, int x)
{
    return push(ch, x);
}
```

```
bool isEmpty(Chan ∗ch)
{
    return ch.back = ch.front;
}
```

```
bool isFull(Chan ∗ch)
{
    return (ch.front − ch.back) % countof ch.buf = 1;
}
```

Figure 3.7: Sequential single-process implementation: channels



```
Chan ∗u, ∗v, ∗w, ∗x, ∗y;
u := mkchan(malloc(sizeof ∗u), 1);
v := mkchan(malloc(sizeof ∗u), 1);
w := mkchan(malloc(sizeof ∗u), 2);
x := mkchan(malloc(sizeof ∗u), 1);
y := mkchan(malloc(sizeof ∗u), 2);

// Processes declared as:
// process A(out u, out v)
// process B(in u, out w, out x)
// process C(in w, out y)
// process D(in x, in y)

Proc ∗a, ∗b, ∗c, ∗d;
a := mkproc(malloc(sizeof ∗a), ..., {OUT, u}, {OUT, v});
b := mkproc(malloc(sizeof ∗b), ..., {IN, u}, {OUT, w}, {OUT, x});
c := mkproc(malloc(sizeof ∗c), ..., {IN, w}, {OUT, y});
d := mkproc(malloc(sizeof ∗d), ..., {IN, x}, {IN, y});
```

Figure 3.8: Sample four-process network

3.2.3 Multiple processes

Let us now build a graph out of our sequential Kahn processes. Our goal should be quite modest for now. The graph will be static: constructed once and unchanging ever after.

From the viewpoint of the host program operating the coroutines, managing multiple processes involves two additional tasks: linking processes together through channels, and choosing what to do each time we are given control, e.g., when *cocall* returns.

Meshing the processes together is simply a matter of first creating the *Chan* objects with *mkchan*, then the different *Proc* objects by calling *mkproc* and passing the appropriate channels as in and out arguments. See Figure 3.8 for an example.

The question of what to do when we have control is in many ways more interesting. To push the overall network toward progress (or completion, in the case of a finite computation), we need data to be produced in channels as they are needed. For half-bound channels, which are only connected to a process on one side, there must be some specific logic, that varies by use case, that provides inputs and consumes outputs. For example, the program could be reading and writing to system file streams. We conveniently assume the presence of the generic *input* and *output* statements that abstract away those details. Input statements return data to be pushed onto an unbound input port; output statements consume items popped from unbound output ports.

For all other channels, both ends should be assigned to some process. It is thus a matter of **scheduling** appropriate coroutines at the right time. A fundamental property of Kahn process networks is their deterministic nature: as shown by Kahn [1974], all choices in terms of what routine to schedule and when lead to compatible streams of values being written to all channels. By compatible, we mean that the streams thus produced are totally ordered in a prefix relation. Some schedules might produce shorter sequences, if, for example, a process enters an infinite loop (which we actually disallow as per the finite transition rule above), or if the operator always chooses to run a same process that is in a waiting state and thus cannot progress further, unless data or space is made available, usually by passing control to others.

3.2.3.1 Compatible schedules

We can give a short concrete reformulation of Kahn's result in the context of our work. Given a Kahn process network, with processes defined as state machines, according to the previous subsection. Process transitions occur in two parts: first the process state is updated by the *step* procedure; then the associated channel operation (written to the reserved cells of the *state* array) is performed. Both actions, which we call start and end half-transitions, happen as part of *cocall*.

We consider **legal** schedules of a given process network: interleaved sequences of alternating process half-transitions, as generated by *cocall* on processes of said network, in any permissible execution. The contract for *cocall*, *chsend* and *chrecv* impose no particular restriction on when they may be called. They simply fail without effect if nothing can be done. Therefore, we may quantify over all arbitrary sequences of invocations. Even though we are in a sequential setting, we still need to assume non-interference from the client. In particular, this ensures that channels are correctly operated, with values being enqueued then dequeued. Let us show the following theorem.

Theorem 1. *If two legal schedules contain the same number of half-transitions for some process p, then:*

- *they have the same process state s for p;*
- *and the same history of produced values $u_0 \ldots u_{k-1}$ (resp. number of consumed values) for every channel u whose producer (resp. consumer) process is p.*

Proof. We proceed by induction on the length of schedules (all legal):

- If the schedules have length zero, then every counts of half-transitions are zero, and all the schedules are the same, so both properties hold.

- Suppose that for some n, the properties hold for every two schedules of length less than n. Let γ be a schedule of length n and γ' another of length n'. Without loss of generality, we assume $n' \leq n$. Let p be some process with the same number of associated half-transitions in both γ and γ'. We recall that half-transitions of a given process are strictly alternating.
  - Suppose the last half-transition of p in γ and γ' is a start half-transition. Since only half-transitions modify the corresponding *state* array, we consider the shortened prefixes that cut before the last (start) half-transition of p in both schedules. Those prefixes have length less than n, thus, by induction, they leave p in the same state. Since *step* is a pure function over states, the new start half-transition produces the same state for p in both γ and γ'.
  - Let u be an output channel of p. Since only end half-transitions of p may produce values in u, we consider the shortened prefixes that cut before the last end half-transition of p that affects u in both schedules. Those prefixes have length less than n, thus, by induction, they leave p in the same state and produce the same history of values on u. The new value added to u is read from the *state* array of p, which is the same by induction. Therefore, it is equal in γ and γ'. Since previous values produced on u are also equal, the new histories of values are equal.
  - Let u be an input channel of p. Since only end half-transitions of p may consume values from u, we consider the shortened prefixes γ_u and γ'_u that cut before the last end half-transition of p that affects u in both schedules. Let k be the index read by that half-transition. Those prefixes have length less than n, thus, by induction, they leave p in the same state and consume the same number of values on u. The prefixes that produce exactly k values on u are even shorter (else, the half-transition could not occur), therefore the values at index k in u in both γ_u and γ'_u are equal. The new value taken from u is written to the *state* array of p, which is also the same in γ and γ'. Therefore, the new states are equal. □

Corollary 1. *If two legal schedules have the same number of half-transitions of both kinds for every process, they are* ***equivalent****, in the sense that their overall shared state is the same. Thus, any transition that can be appended to one can be equally appended to the other to yield two new equivalent extended schedules.*

Corollary 2. *Given a legal schedule γ, if there exists a legal schedule γ' and a process with more half-transitions in γ', then there is at least one half-transition that can be appended to γ to yield a longer schedule.*

Proof. There are two cases:

- If there exists a process p that has more start half-transitions in γ' than in γ, then appending a start half-transition of p to γ is always legal.

- Otherwise, all processes with more half-transitions in γ' only have one more end half-transition. Let p be the process associated with the earliest such end half-transition α, and γ'_α the prefix of γ' that excludes every action from α (inclusive) onward. Suppose the added end half-transition consumes at index k in some channel u. The state of p and the number of consumed values on u in γ and γ'_α are the same since both schedules have the same number of corresponding half-transitions for p. Since γ'_α contains less half-transitions than γ for every process, in particular, the first k of u must have been produced in γ, and α can be appended to γ to consume the value at index k. □

Lemma 1. *Given a bounded Kahn process network, the same network with larger bounds on some of its channels produces compatible results. For every execution of the tighter network (with stricter bounds), there exists an execution of the looser one that produces at least as many values on every channel.*

Proof. Any schedule of the tighter network is also a valid schedule of the looser one, in which the scheduler could have taken other decisions with regard to production on channels where space was available but decided not to. This proves the second point. By Theorem 1, all those schedules are therefore compatible with any other schedule allowed by the looser network. □

The above lemma states that increasing the bounds on channels of a given network may only lead to more output being produced. Up to this point, we have considered general bounded Kahn process networks. In the rest of this thesis, however, we will only be interested in programs in which the boundedness of channels merely acts as a resource constraint and does not otherwise affect the outcome. Intuitively, we focus on programs with sufficiently large buffers. For a process network to qualify, its results should be the same as would be produced by the same network where every channel is unbounded. That is, if there exists a schedule of the unbounded network that produces a certain number of values, then there must exist a schedule of the bounded network that does the same. In particular, this excludes programs that produce more values the longer the channels we give them. If precise acknowledgment of consumed values is needed, we must add explicit reverse channels to this effect, instead of relying on the implicit feedback provided by bounded pushes.

The reason we require such implicit channel sizes has to do with performance. We generally want the programmer to specify only a lower bound on each channel, so that the implementation may decide to round it up however it likes, which will have important practical applications later on.

3.2.3.2 Maximal progress

In general, we want schedulers that guarantee **maximal progress** conditional on the root process. That is, at any point in time, if there exist a schedule that leads to more root transitions than the current schedule, then the maximal-progress scheduler eventually makes progress with probability one.[1] As a baseline, we should note that any scheduler that picks what next action to attempt randomly satisfies the criterion, provided it is minimally fair. A scheduler is **minimally fair** if there is a constant lower bound on the probability it will choose any given item in a round.

Lemma 2. *Any minimally fair scheduler makes maximal progress.*

Proof. If there is a schedule with a longer transition sequence for the root process, then Corollary 2 tells us that there exists a target process p that, if selected, makes progress. Since it is minimally fair, each attempt has at least some constant probability of choosing p. Therefore, it will sooner or later pick p, with probability one. □

This method is not, however, very efficient as many of its actions are wasted on *cocall* invocations that initiate no new transitions. Fortunately, we have defined the interface such that *cocall* returns just enough information to tell the environment what it needs to continue (i.e., data or space in a specific buffer). Therefore, a very simple yet effective strategy for a sequential interpreter is **demand-driven scheduling**, one of the approaches discussed in [Kahn, 1974].

In such a scheduler, shown in Figure 3.9, the scheduling function, *demand*, takes two process arguments. The first one is the source process that we wish to run. The second, the target process, specifies when we should stop. The operator invokes *cocall* on the source process repeatedly until it produces something on the channel bound to the target process—which is presumably waiting on it. The current work may itself require other processes to run, and thus calls *demand* recursively. If needed, then input or output by the environment is performed.

Lemma 3. *If* demand *is recursively called with a source argument p that has already been passed to a pending invocation (on the stack), then it loops forever. The schedule generated at the second call site contains the same number of half-transitions of p as the one at the point where the first invocation recursed.*

Proof. Suppose *demand* is called recursively with a source argument p that already appears on the stack. Consider the first such occurrence. This implies that all source processes on the stack except p are distinct. Morever, p is in a state that requires progress on some channel u (in some direction). That progress can only be achieved by running the opposite process q (by the single-producer single-consumer rule). Since q appears at most once in the sequence of *demand* invocations between the two occurrences of p, it must be the immediate recursive call following the first of p. Since that call did not return, it did not complete its target condition, thus did not affect u. Therefore, the second occurrence of p cannot proceed any further than the first. No new half-transition has been introduced for p, and the loop repeats identically. The same argument can be made for every following process we encounter in the ensuing cycle. Each new cycle introduces one new occurrence of every process in the loop, which similarly does not return, thus

[1] Note that we do not require progress to be made on the root process specifically, simply that it acts as a global termination condition.

```
// Returns the adjacent process following the specified link,
// or null if the channel is unbound on that side.
Proc *adjacent(Proc *, Chan *);

// Returns the argument of the process bound to some channel.
Arg arg(Proc *, Chan *);

// Runs the first process just enough times to be able
// to yield to the second one.
void demand(Proc *p, Proc *r)
{
    for (;;) {
        var ok, ch := cocall(p);
        if (ok and ch = null)
            return;
        var q := adjacent(p, ch);
        if (ok) {
            // Is the demand satisfied?
            if (q = r)
                return;
        } else {
            if (q ≠ null)
                demand(q, p);
            else {
                switch (arg(p, ch).kind) {
                case IN:
                    var x := input(ch);
                    chsend(ch, x);
                    break;
                case OUT:
                    var _, x := chrecv(ch);
                    output(ch, x);
                default:
                    abort();
                }
            }
        }
    }
}
```

Figure 3.9: Sequential demand-driven scheduler

does not affect the required channel. Finally, no new half-transition is introduced from the first repetition onward. □

Lemma 4. *The demand-driven scheduler makes maximal progress.*

Proof. Suppose the demand-driven scheduler enters an infinite loop and stops making progress. Since we consider only finite graphs, this only occurs if *demand* recurses indefinitely, as in Lemma 3. Moreover, there is a finite set W of processes covered by this loop, including the root process. Equivalently, it is the set of processes present as arguments to *demand* invocations on the stack. Let γ be the current schedule.

Suppose there is a legal schedule γ' that has a longer half-transition sequence for the root process. Let α be the earliest half-transition in γ' but not in γ, of a process p belonging to W. Since *demand* invokes *cocall* on every process it touches, the last half-transition for each is a start half-transition, thus α cannot also be a start half-transition. Let u be the channel that is operated on by α.

Let γ'_α be the prefix of γ' up to but excluding α. Since γ'_α and γ have the same number of half-transitions for all processes in W, all related channels have the same histories of produced and consumed values, including u. Let q be the other process bound to u, that is not p.

We look at a pending invocation of *demand* (on the stack) that has p as first argument. Since p is in W and its state mandates an operation on u, this invocation must have generated a recursive call to *demand* with argument q, which did not return, by hypothesis. Thus, it did not match its target, and u was not affected. By Lemma 3, we know that no following invocation involving q could have added any new half-transition. Therefore, the state of u in γ is the same as it was when p first recursed. The half-transition α cannot be applied to such a state, hence not to γ' either. We have a contradiction. □

Lastly, in preparation of things to come, we would like to point out that any scheduler can be made minimally fair, hence achieving maximal progress, by interleaving its choices with a random scheduler at bounded intervals; e.g., pick one process at random every k clever choices for some k. This can be used advantageously by choosing k such that those random rounds are amortized by the number of processes in the network, in combination with unsafe schedulers that do not have a maximal progress guarantee. While not very interesting for a sequential algorithm, we will use this method to turn lock-free schedulers that are non-maximal in the presence of faults into minimally fair schedulers. There is a gain to be had in both practical efficiency and simplicity of design.

Yet other variations and heuristics are also possible. We shall, however, be content with the above demonstration of sequential scheduling, which paves the way for the next important topic: reconfiguration.

3.2.4 Recursive processes in the sequential world

To complete our sequential Kahn process network implementation, we need to support reconfiguration: the ability for one process to split itself into several children connected by channels. In a Kahn process network, reconfiguration happens at most once per process, as it effectively replaces the parent with its children, much like *execve* replaces a Unix process with another.

Put another way, a reconfigured process delegates all further actions it is allowed to take to its subprocesses. We can think of *reconfigure* as recursive interpretation. As a statement, it combines the initialization of a network of child processes, followed by a transition to a final state that loops while scheduling those child coroutines, as we have seen in the previous subsection. To the external observer, it should be invisible: it seems like the parent process could be carrying out the exact same work in sequence instead of running a nested interpreter. But could it really?

Unfortunately, there is a certain mismatch between single processes and networks of those: the former behave sequentially and deterministically; the latter allow for different schedules, as we have seen, and are therefore non-deterministic.

In particular, central to the Kahn design, one process can only wait on one channel at a time. Once reconfigured, however, the subgraph is inherently concurrent and different child processes might have different needs. Since the the inner scheduler has no knowledge of coroutines outside of the enclosing parent scope, it cannot decide between requests for different channels. If child A requires data from channel u while child B wants to write to channel v, and both channels are bound only on one side within the subgraph, how do we know which one to ask for and return to the outer environment as the result of *cocall*? If we ask for the wrong one, the parent scheduler might just not be able to satisfy the request because of dependencies unseen by the deeper level. Perhaps u and v must alternate, or maybe u can only be read after five values have been consumed from v. A perfectly nested scheduler cannot know the answer to those questions.

This shows that, while *cocall* might be suitable for single processes, we need a different interface to deal with networks. We define a new function *copoll* that borrows much from *cocall*, except it orchestrates the execution of a whole network rooted at a given process. Instead of asking to run a specific process, the caller gives *copoll* a set of channels for which it is responsible. In a way, it is similar to multiplexing system calls such as *select* and *poll* on Unix, or *WaitForMultipleObjects* on Windows, albeit with different timing priorities. The *copoll* function does not return for each individual movement within the network. Instead, it only yields control to its caller when it registers a request for more data or space on one of the provided channels, or on special conditions such as termination. This is in contrast to *select*, which attempts to return as soon as any operation is possible on any monitored file descriptor. Our *copoll* procedure is rather lazy and only gives control back whenever intervention by the environment is required. This does not change whether progress is made; it simply affects which side advances first. In our case, Kahn processes are given the preference.

We demonstrate an implementation of *copoll* and *cocall* in Figures 3.10 to 3.12, where a single topmost *copoll* invocation handles any level of nested subgraphs. Alternatively, we note that a recursive implementation should be possible, by breaking the single-process abstraction of *cocall*, and considering each node as a potential subgraph. Recursive execution would then be handled by *copoll* itself, making for a network of networks. We feel such an arrangement brings little to the table, since the graph of subgraphs

```
struct Conf {
    Proc procs[∗];
    Chan chans[∗];
};

struct Proc {
    bool step(Arg[∗], int[∗]);
    Arg args[∗];
    int state[∗];
    Conf ∗reconfigure(Arg[∗], int[∗]);
    Conf ∗conf;
    bool finished;
};
```

Figure 3.10: Single-scheduler reconfiguration: declarations (changes only)

could really just be handled as one big network instead, which is both simpler and more efficient.[2]

We define global termination as the state of every process in the network having finished. An alternative would be to have one child inherit the root process property, which makes it possible to reuse the demand-driven scheduler almost unchanged, as suggested in [Kahn, 1974]. While it might seem appealing, as we will see, the demand-driven scheduler is not a good fit in a concurrent context. Plus, such a design complicates having multiple parallel outputs handled by different processes—care must be taken when picking a root process that indeed finishes after all others. Therefore, we opt here for the more straightforward solution.

The implementation relies on the existence of a scheduler routine, *schedule*, that picks processes to execute, with the same caveats already discussed in the previous section (e.g., maximal progress). Fortunately, any scheduler (including all those in the previous subsection) that only navigates the graph from vertex to vertex (each representing a process) through channel edges remains conveniently compatible with this new organization. We postpone further discussions of process scheduling to Section 4.1. Thus our sequential Kahn process network implementation can be considered complete, with full support for networks rather than single processes.

[2] Connecting multiple networks run by different interpreters would be a valid reason to have *copoll* handle only part of a larger graph. While we believe this should be possible in a sequential world using nothing more than the *copoll* interface defined here, the situation is a lot more complex in the concurrent case, and is overall beyond the scope of our present work.

```
// Number of live processes left to run.
int nproc;

// Returns a process to execute, or null if none can run.
Proc ∗schedule(Proc ∗, bool w[]);

int copoll(Proc ∗p, bool w[])
{
    for (var i := 0; i < countof p.args; ++i) {
        if (w[i]) {
            var arg := p.args[i];
            if (arg.kind = IN and isEmpty(arg.ch) or
                  arg.kind = OUT and isFull(arg.ch)) {
                return i;
            }
        }
    }
    while (nproc > 0) {
        var q := schedule(p, w);
        if (q = null) {
            // Deadlock.
            return −2;
        }
        var ok, ch := cocall(q);
        if (not ok) {
            for (var i := 0; i < countof w; ++i) {
                if (w[i] and p.args[i] = ch)
                    return i;
            }
        }
    }
    // Global termination.
    return −1;
}
```

Figure 3.11: Single-scheduler reconfiguration: *copoll*

```
(bool, Chan ∗) cocall(Proc ∗p)
{
    if (p.finished)
        return true, null;
    if ((p.state[0] = −1 or p.state[1] = −1) and
          p.step(p.args, p.state)) {
        p.finished := true;
        −−nproc;
        if (p.reconfigure ≠ null) {
            p.conf := p.reconfigure(p.args, p.state);
            nproc +:= countof p.conf.procs;
        }
        return true, null;
    } else {
        // Usual channel input/output handling.
        ...
    }
}
```

Figure 3.12: Single-scheduler reconfiguration: *cocall* (changes only)

3.3 Concurrent interpretation

3.3.1 An interpreter shared object

From our sequential implementation in the previous section, we now shape a concurrent lock-free Kahn process interpreter: one that makes use of multiple threads. In doing so, our objective is evidently to exploit the inherent potential for parallelism in Kahn process networks—an interesting case of disjoint-access parallelism.

Before we delve into further implementation discussions, there is one question that needs to be addressed: what is it that will run in parallel? In other words, what is the protocol that we intend to allow?

Instead of a procedure, we constructed our sequential Kahn process interpreter as a stateful object, which is ideal as a foundation for the sequential specification of a future linearizable concurrent data structure. It has three top-level methods: *copoll* (which supersedes *cocall*), *chrecv* and *chsend*. As we have seen, these cover three classes of observable behaviors: data transfers from and to the network, plus termination.

Crucially, concurrent invocations of *copoll* should be allowed and, as a rule of thumb, they should take care of different parts of the network, so as to be able to run processes in parallel without competing for too many shared resources. Due to its dependency on a black-box scheduler, in most cases, where multiple process choices can be made, the semantics of *copoll* is naturally non-deterministic, from the viewpoint of its caller.

The main point of contention is with regard to the channel methods: should we allow multiple threads to read from or write to the same unbound channel port? There are mainly two alternatives:

1. Disallow concurrent client accesses to the same channel: the client must guarantee that all invocations *chrecv* or *chsend* to a same channel object be exclusive. This does not apply to transfers made by the run-time library, which, as we have explained, are necessarily concurrent if we want to attain lock freedom.

2. Allow concurrent client accesses to the same channel that linearize to a first-in first-out queue. Each thread gets to read or write one item of the buffer in some sequence consistent with how it appears to Kahn processes: e.g., client threads read out values in the order they were added to the channel by the Kahn program.

Given the signatures of the *chrecv* and *chsend* methods defined above, there is hardly any other possible interpretation. We could however imagine a more involved interface with indexed accesses. This would add much complexity and depth to the question. Are out-of-order accesses legal? Do we permit concurrent stores of different values at the same index? What about those that assign the same value to the same spot?

In this work, however, we settle for the simplest dynamic: exclusive access to a same channel. We justify this choice with two arguments: one conceptual, and one practical.

On a conceptual level, we would like to view the entire system as blending Kahn processes with system threads to fabricate a composite network made of two types of **nodes**:

Soft nodes high-level Kahn processes, that are lightweight, cooperative, and guaranteed non-blocking by our run-time library;

Hard nodes low-level system threads, that are heavyweight, typically few in number, and may carry out any duty, including blocking tasks[3] that require waiting on external input or output conditions, e.g., from disks or sensors.

This system would still follow the Kahn process network philosophy—or at least appear to do so. Most importantly, the single-producer single-consumer rule would apply: meaning there is at most one node, either soft or hard, on each end of a channel. Hence, the exclusive access restriction on the use of *chrecv* and *chsend* on a same channel.

From a practical angle, this constraint also makes most sense, as it permits direct reuse of part of the internal channel implementation, which is not suited for use as a multi-producer multi-consumer queue. Indeed, even though we must allow contention between multiple readers or writers in order to avoid deadly waits, we surely want to take advantage of the main feature of Kahn process networks: deterministic computations,

[3] They lie on the other side of our lock-free criterion, which we believe to be totally reasonable. Such treatment would typically be reserved for things that cannot be accomplished satisfyingly within a normal Kahn process, such as reading from a socket. Even if multiplexing calls such as *select* exist, the actual ensuing input operation can usually not be repeated, which does not lend itself easily to lock freedom, to say the least.

which is implied by the pureness of the *step* function as specified in the previous section. Under this hypothesis, values written to a given channel at a given position by concurrent threads interpreting the same Kahn program are always equal: a fact we can leverage in our algorithms. This, however, also makes our channel objects unfit for use as general multi-producer multi-consumer queues, as would be required if we were to allow arbitrary concurrent invocations of *chrecv* and *chsend*.

It should also be noted that this protocol does not prevent the Kahn process network from communicating with its environment through as many channels as desirable. Each channel may be bound to a different hard node—a thread. In addition, there may also be threads that do not perform any input or output operation and simply call *copoll*. On a semantic level, they may do so in order to observe termination of the process network. In practice, they help execute spare soft processes, whose side effects on channels are seen by hard nodes. We call any thread that contributes *copoll* instances a **worker thread** (or simply **worker**). Taken together, method invocations emitted by all threads interacting with the interpreter shared object, assuming they follow the above protocol, should linearize to the sequential "multi-filter" interpreter described the previous section.

3.3.2 States and transitions in a concurrent world

As a preliminary to data structures and algorithms, let us first discuss some vital characteristics of what we are trying to implement.

As we have seen, internally, the Kahn process interpreter works in steps called transitions, which can themselves be divided into a start and an end half-transitions. The former updates a process state, while the latter performs (internal) input or output on a channel.

Since we wish for disjoint-access parallelism on unrelated transitions, different elements of the graph—such as distinct processes or channels—should be represented and updated separately. This brings the fundamental issue of multi-word algorithms, introduced in the last chapter: safely modifying two locations at once. Those two locations are one for the process, and another for the channel.

Obviously, the process and its channels lived in a single chunk of memory, we could swap it in a single operation. But what about the other processes bound to those same channels then? They would need to be crammed into the very same chunk. Or maybe they could have their own version of the channel objects? But then how do we synchronize those? We are back with the multi-word problem.

Fortunately, Kahn process networks also have very nice properties, most notably determinism, which we can leverage to build better, leaner algorithms.

First of all, our update operations all touch exactly two locations and have a natural ordering: start followed by end half-transition. Moreover, processes are sequential and deterministic, so at any one point in time, there is at most one transition in flight—the most recent—that is legitimate, while all others have already expired, even if their host threads might not have realized it yet.

Lastly, producer and consumer on a channel operate on opposite ends of the buffer, which means that—depending on the representation of channels—some transitions that

```
‘T *safeRead(‘T **);
void beginReadSection(‘T *);
void endReadSection(‘T *);
bool inReadSection(‘T *);

// To make matters simpler, every integer type is int;
// this does not match standard C, where alloc functions
// take size_t arguments.
void *malloc(int);
void *calloc(int, int);
void aalloc(int, int)[*]
void destroy(‘T *, void (*)(‘T *));
void collect();
void free(void *);
void freex(void *);
```

Figure 3.13: Memory management interface

overlap might in fact update different locations altogether.

3.4 Memory management and ABA prevention

As the basis for all of the following algorithms, we assume the existence of a global lock-free memory management and ABA-prevention system, as described in Sections 2.6 and 2.7. We do not specify an exact implementation, although differences in the expressiveness of the interface, and their impact on our algorithms, will be noted. Furthermore, we will be precise as to where to place the necessary read barriers in our Kahn process interpreter code, in order to enable any such system to work. Thus, our algorithms do not require automatic garbage collection.

Before turning to our new data structures, we briefly describe the interface to the lock-free memory management system. For more information, including a full explanation of how to implement one of those (or several variants thereof), please refer to the previous chapter.

The memory management system exposes four functions that cater to read sections, and several functions that operate the per-thread quarantine and (manual) garbage collection mechanism. Signatures for all these functions are shown in Figure 3.13, and explained below.

safeRead Safely loads a block pointer b stored at the given shared location a. Upon return, we are guaranteed that b is locked by a read section as if *beginReadSection* had been called on it atomically as its value was read.

beginReadSection Unconditionally enters a read section on the specified block. It will

```
// aalloc is just a convenience function.
void aalloc(int n, int size)[∗]
{
   var p := calloc(n, size);
   return (void[∗])(n, p);
}
```

Figure 3.14: Memory management: *aalloc*

not be collected by the memory allocator (see *collect* below) before we pass it to *endReadSection*.

endReadSection Leaves a read section started with *safeRead* or *beginReadSection*. Note that read sections on a same block do not nest.

inReadSection Returns true if the current thread is within a read section for the specified block. We never use this call in actual code; only in assertions to establish (sequential) preconditions.

alloc Family of functions that return fresh blocks. The *malloc* function returns a block of the requested size or more. The *calloc* function returns a block of size the product of its arguments—the first argument specifies the number of elements in the allocated array and the second one gives the size of one element. As a reminder, in C, *calloc* zeroes out every bits of the returned region, but *malloc* does not. Finally, the *aalloc* function is defined in Figure 3.14.

destroy Sends the given block, to which the calling thread must have exclusive write access (e.g., by detaching it from the shared data structure in a way that makes it unavailable to other writers), to the quarantine, along with a destructor function to run when there are no more readers.

collect Scans the quarantine looking for blocks that can be safely freed. For each such block found, call its destructor function, which should call *free* on the chunk itself and any wholly owned chunks.

free Marks a block as clear for reallocation by *alloc*. It must be exclusively held by the caller (e.g., designated as such by *collect*); therefore, *free* should normally only be called from within a destructor. We use *freex* when outside of *collect*, e.g., for chunks allocated locally and not (yet) swapped into any shared data structure.

To help with initialization cases, the *inReadSection*, *endReadSection*, *destroy*, *free* and *freex* functions accept null pointer arguments, and produce no effect when so called. The *safeReread* function behaves like *safeRead* if passed a null second argument.

In the following proofs, we generally do not pay much attention to the read-section management routines, except for confirming that a specific use of a pointer acts as if it were unique. This is in large part due to the fact that the memory reclamation system offers a different interface to most other structures. Most importantly, memory addresses only serve as proxies to the blocks they represent; in particular reallocated blocks are semantically distinct entities, while two otherwise identical histories except for block addresses should be considered equivalent. Abstractly, the memory system has two modes of operation.

- Under normal circumstances, when the system uses no more than the available space, and every thread only uses up to its allotted amount of read-protected memory, the allocator behaves as a lock-free source of fresh blocks, albeit with non-deterministic addresses. Very importantly, in this scenario, individual read sections taken by other threads are invisible. Therefore, they act as if they were purely local operations, and for the most part can be ignored when studying linearizable behaviors.

- However, should one of the two above conditions fail, then it may become possible for clients to observe foreign read sections, due to their direct influence on the quantity of memory available.

This being the case, all of our proofs will assume that both the heap and read-section limits are appropriately sized for the application, so that memory management operations appear transparent, and allocations never fail.

3.4.1 Grouped block reclamation

An important optimization that was only mentioned in passing in the previous chapter is the grouping of multiple actual chunks into under a single master block that acts as the unique protected resource, with regard to memory reclamation.

There are many scenarios where read sections can be elided, depending on the arrangement of the blocks. We are interested in a very restricted use case that we call **bins**, after the name sometimes taken to designate the equivalent grouped allocation in sequential programs. The principle is simple: blocks can be grouped together into a bin, and more blocks can be added to the bin, but never removed, unless the entire bin is freed. A bin is always represented by its original chunk (or **root**), which is the only one directly visible to the memory management system, with regard to the shared–exclusive life cycle. All other blocks in the bin share its fate. In particular, shared access to any block in a bin requires taking a read section on the root, which must be held for the duration of the operations, as usual.

The only implementation of the bin concept that we will use is the **tree bin**, in which chunks are organized as nodes in a tree. The whole structure is represented by its sole root—as a normal tree would be in a sequential world—and no location external to the object may point to one of its blocks, except the root. Consequently, swapping the root chunk in and out of shared data structures acts as an exclusion mechanism.

Sample code for a generic tree bin is provided in Figure 3.15. In practice, we will want to specialize the layout and branch handling code to fit a fine-tuned existing structure. In particular, having leaves also count as tree nodes is both wasteful and cumbersome.

Each node of a tree bin has a number of designated branch fields that hold pointers to other nodes, or null for a leaf. Each of those branch fields may be swapped exactly once, from null to point to some additional block that is thus added to the bin. Since branch fields cannot be modified again, those blocks never leave the bin thereafter.

Reclaiming the entire tree bin is just a matter of waiting on its root and traversing the tree sequentially, as we would a normal tree. Adding nodes is done by modifying one of the existing blocks in the bin, which requires a read section on the root. Therefore, if all read sections expire while we have exclusivity over the tree bin, no more chunks can be added, and freeing all those that are currently linked is safe (no external pointer to any block and no live read section) and does not leak memory.

3.5 Overview of the concurrent interpreter

With memory management well-understood, we are ready to start with the construction of the concurrent interpreter itself.

The following sections propose an in-depth bottom-up construction of the concurrent interpreter, as we did for the sequential version, only with (a lot) more intermediate steps. At every stage, we prove linearizability of the components presented, for easy reuse at the next level.

Sections 3.6 and 3.7 present the two data structures at the lowest layer: monotonic buffers in Section 3.6, and macro queues in Section 3.7. The two must be used in tandem, according to the monotonic-block replacement technique introduced in the previous chapter, to form a full channel that connects two Kahn processes. As we will see, this channel object is unlike the usual lock-free queues designed for thread-to-thread communication. It specifically supports non-blocking transitions, as defined above, and, in exchange, takes advantage of the single-producer single-consumer nature of Kahn processes.

Then, in Section 3.8, we build upon those combined channels to create a process graph, by adding process nodes, and methods to access and update both processes and channels in a consistent way. As regards the representation of process networks, we present a flexible three-layered design:

- an authoritative consensus-based shared base;
- an efficient monotonic layer where most communication operations happen; it permits exchanges between Kahn processes controlled by different threads using only load and store instructions;
- a local layer, where threads maintain cached views of the process states.

Section 3.9 implements the missing transitions—without which there would be no action—on top of processes and channels, and finally in Section 3.10, we conclude by

```
// Embed as the first member of a custom structure.
struct Tree {
    Tree ∗root;
    Tree ∗br[∗];
};
```

```
// Call this after allocating a new chunk for a tree node.
Tree ∗makeTree(Tree ∗t, int nbr, Tree ∗∗pbr)
{
    t.root := t;
    t.br := (Tree ∗[∗])(nbr, pbr);
    return t;
}
```

```
bool growBranch(Tree ∗t, int bi, Tree ∗new)
{
    assert(inReadSection(t.root));
    assert(bi ≥ 0 and bi < countof t.br);
    assert(new.root = new);
    new.root := t.root;
    if (cas(&t.br[bi], null, new))
        return true;
    new.root := new;
    return false;
}
```

```
// Pass this to destroy on the root.
void freeTree(Tree ∗p)
{
    for (var i := 0; i < countof t.br; ++i) {
        if (t.br[i] ≠ null)
            freeTree(t.br[i]);
    }
    free(t.br[i]);
}
```

Figure 3.15: Tree bin example

writing a concurrent version of *copoll* and proving a correspondence with sequential schedules.

3.6 Monotonic channels for finite closed programs

Let us begin our exploration with a very simplified version of the problem, by considering only finite closed forms of process networks, that receive no input and terminate in a fixed amount of time.

As a starting point, we want to give a word-monotonic implementation of such an object. The goal of the closed Kahn program is to fill one or several output buffers with values from its computation. The fully redundant but nonetheless lock-free[4] first solution should be to run multiple sequential interpreters in parallel and simply write the results to the same output buffers. Due to determinism (Theorem 1), we know that whatever values get out will be equal, which means we do not need to worry about conflicts even when running different schedulers or random ones.

Naturally, we wish to do better than that. In particular, we seek the much-touted disjoint-access parallelism property: we would like for different threads to partake in the execution of our single network, seamlessly handling disjoint parts of the graph in parallel.

What if we just implement all channels as shared buffers? Threads can then simply execute processes to fill the individual buffers. The same determinism applies to those intermediate values as they do to the final output. Failing or delayed threads can be replaced simply by peers picking up their assigned processes and restarting them from the beginning. This will certainly regenerate and overwrite parts of the associated buffers that had already been filled, but is not harmful (and should be infrequent, anyway).

However, we do face another problem: synchronization between producers and consumers. How can one thread tell how much content has been produced in a given channel controlled by another thread?

Fortunately, we need look no further than the saturating counter of the previous chapter. The basic trick is to have every word-sized modification move the overall state forward in an non-ambiguous manner: values never repeat at any location. If we view the entire collection of shared variables as a large tuple $(w_0, \dots, w_{n-1})$ (one entry for each word), then the set of possible values for our system must be arranged in a partially ordered set, where the overall order is compatible with the separate orders over individual words that comprise the tuple: $\forall w, w',\ \forall k,\ w_k \leq w'_k \implies w \leq w'$.

There are many ways to achieve this, with various trade-offs. We can use an index variable that keeps track of how many values have been pushed in a given channel. The consumer then only needs to poll for increased index values. This solution is trivially word-monotonic. However, it suffers from a common caveat: it forces a costly read-modify-write operation onto the producer, in order to deal with delayed and repeated processes without repeating values.

[4] This is not at all surprising, given that lock freedom is a fault tolerance property, and strategy zero for any fault tolerance system is pure redundancy.

```
bool monoPush(atomic int buf[∗], int i, int x)
{
   assert(i ≥ 0 and i < countof buf / 3);
   if (i > 0 and not buf[(i − 1) ∗ 3 + 1])
      return false;
   buf[i ∗ 3] := x;
   buf[i ∗ 3 + 1] := true;
   return true;
}
```

Figure 3.16: Monotonic buffer: *monoPush*

```
(bool, int) monoPop(atomic int buf[∗], int i)
{
   assert(i ≥ 0 and i < countof buf / 3);
   if (i > 0 and not buf[(i − 1) ∗ 3 + 2])
      return false;
   if (not buf[i ∗ 3 + 1])
      return false;
   var x := buf[i ∗ 3];
   buf[i ∗ 3 + 2] := true;
   return true, x;
}
```

Figure 3.17: Monotonic buffer: *monoPop*

Figures 3.16 and 3.17 demonstrate another idea, using flags, as follows. We replace integer buffers with arrays of triples:

1. the value written out;
2. a write flag that goes from false to true to indicate the presence (production) of a value;
3. a read flag that goes from false to true to indicate that it has been consumed.

Again, the whole thing remains word-monotonic, and we have eliminated our use of complex instructions. As a rule of thumb, we will always try to keep expensive atomic primitives such as compare-and-swap to a minimum, preferably completely out of the common "fast" path, if possible, or else, at least amortized over several cheaper operations.

Lemma 5. *Assuming the client calls* monoPush *always with the same pairs of index–value arguments, the* monoPush *and* monoPop *functions are linearizable.*[5]

Proof. Successful invocations of the *monoPush* function linearize when they set the write flag. Analogously, the *monoPop* function linearizes when setting the read flag. Failed invocations linearize on the test they fail.

The abstract value of the monotonic buffer is a list of values that have been written, and a count of how many have been consumed. Values are considered to exist only when their write flag is set, so the abstract list of values only changes on line 7 in *monoPush*; on the consumer side, change occurs when the read flag is set, also on line 9. We then verify that: on the one hand, when run sequentially, the behavior of methods depends only on this abstract value; on the other hand, by induction on linearization points in an arbitrary history, each invocation behaves exactly like the sequential equivalent applied to the abstract state at the point of linearization. □

3.6.1 Replacing buffers

The main limitation of this approach should be evident, however. It does not handle arbitrary Kahn programs, since the buffers need to be dimensioned just large enough to hold the entire sequence of values that will ever be produced by each process, instead of just the lower bound necessary to hold temporary values in channels as they transit from one point to another.

Thus, we ought to ask ourselves: what would be needed to permit buffer reuse? Going down the same path we took in the last chapter, the next logical step would be to upgrade to a chained-block structure. Following this method, monotonic sequences are to be divided into cycles, separated by a reset operation. This reset operation replaces full blocks (that have reached the top value) with fresh ones in the bottom state. In our case, it means swapping full buffers for empty ones.

[5] When we do not specify the sequential target of such linearization, we mean, by convention, linearizable to the same function run sequentially.

We thus set forth on a quest to create a block-based channel algorithm that supports wrapping around, by swapping blocks. At the same time, it must expose an interface compatible with our Kahn process network needs.

At first glance, our channels may look very similar to standard first-in first-out queues, that allow multiple threads to enqueue and dequeue elements in a linearizable way. However, they differ in a fundamental way: they are used to simulate single-producer single-consumer channels in the higher-level Kahn language. This is apparent through the above dependency on process states.

Let us look at the producer role, for example. Although it is true that multiple threads will be attempting to write into a same channel, they will all do so while "impersonating" (interpreting) the same Kahn process: the single producer bound to that channel. That process is sequential, and at each step performs at most one operation on a given channel, which must complete before the next step can be computed and another channel operation considered. In other words, channel operations are interleaved with, and dependent upon, particular process states.

Thus proper channel implementation in a Kahn process interpreter starts with a close examination of states. To begin, what is a process state? Until now, our monotonic implementation does not actually specify a representation for process states. In fact, quite the opposite: we have explicitly seen that in order to assist another thread, its processes have to be repeated from the beginning, with some redundancy in the output. Any intermediate state is thus known only to the worker thread and not persisted as a value in shared memory.

This only works because the purely monotonic implementation preserves all values inserted into channels since the beginning of time and forever. Thus, any process can be rebooted without fear, for any necessary data it may have to read is still present and can still be found in the same channels where they were first appeared. Looking back, we notice that although we defined read flags for our data, they are actually optional: nobody needs to know when data has been consumed, if everything is only filled once. Of course, we did so in order to introduce the present block replacement problem.

When block reuse is added to the game, things become more interesting with regard to states. Once a block is swapped out, the intent is to recycle the associated memory. Previous values are lost. Therefore, processes need to persist their states to shared memory, lest they should be lost if the host thread halts. And unlike before, without prior consumed values to rely on, they cannot be restarted from the beginning. At least, they cannot be restarted in isolation.

We could, of course, imagine an implementation that also reapplies any further process, deeper in the dependency chain, as necessary to reproduce those already consumed values. This strategy, however, fails in the presence of possibly infinite streams of inputs that cannot be stored. Therefore, an interpreter supporting infinitely running Kahn process networks needs process states to commit to shared memory, from time to time, at **checkpoints**. Those checkpoints need to be at least as frequent as channel block changes, to make up for the lost information discarded along with the previous chunks.

In concrete terms, we associate with each process a single variable that points to a block representing its state at the last checkpoint. It is simply updated with a compare-and-swap statement whenever needed, so as to establish consensus between potential competing worker threads.

3.6.2 A non-working chained-block strategy

We start by examining a natural yet misleading idea, which goes something like the following: “This is a monotonic-block algorithm, and we already have one chunk per channel. Therefore, we need to replace that unique chunk every time it is saturated.”

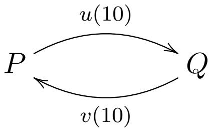
This sounds simple enough, however, it suffers from one major flaw. As matter of course, we want to keep disjoint-access parallelism, that is, parallelism between threads working on different parts of the Kahn process graph. Therefore, we must update each channel separately. Even so, there remains the issue of producer–consumer decoupling. In our current progress scheme, buffers are updated monotonically until every word is produced and then consumed. If we wait until this top state is reached to swap out the block, however, we are breaking bounded channel semantics. Indeed, when we are at the end of a chunk, we must wait for every triple to be read before pushing in contents into a new block. This does not fare well in many configurations that may require more production before the last few values are cleared to make way for a chunk replacement that will thus never come.

To illustrate this, imagine the following example, depicted in Figure 3.18. Two processes P and Q are mutually connected by one channel in each direction, and take turn being producer and consumer. Each channel can hold 10 elements, and the two processes want to exchange 15 values. P pushes 15 times, then pops 15 times. Q on the other hand starts by reading 5 values, then proceeds to push 15 and pop the remaining 10. In a correct implementation, a possible schedule is: P pushes 10; Q pops 5; Q pushes 10; P pushes 5; P pops 10; Q pushes 5; P pops 5; Q pops 10. However, if we do not allow push operations to occur while the buffer is half-read, then after Q first pushes 10, P still needs to push 5, yet Q is not popping anymore and the buffer cannot be cycled: we have a deadlock.

The central issue with this naive approach thus stems from the fact that a monotonic buffer of length L_{B} is only able to accurately represent all the states of a normal queue of length $L \leq L_{\mathrm{B}}$ up until it is entirely filled with (read or unread) items and there are less than L elements left to read. More precisely, let p be the position of the highest set write flag and c the position of the highest set read flag; we consider a monotonic buffer to be in a **degenerate state** whenever:

$$p = L_{\mathrm{B}} \wedge c > L_{\mathrm{B}} - L$$

At that point, no new data can be pushed into the buffer, yet the queue contains less than L elements: it should not be full. In a normal ring buffer, whenever the structure contains less than its full length L, new push operations are allowed. Thus, the two differ in their semantics. This is undesirable, since it may introduce unwanted deadlocks, as we have just seen in Figure 3.18.



```
process P(in v, out u)
{
    int i;
    int x;
    i := 0;
    while (i < 15) {
        u ← i;
        i := i + 1;
    }
    i := 0;
    while (i < 15) {
        x ← v;
        i := i + 1;
    }
}

process Q(in u, out v)
{
    int i;
    int x;
    i := 0;
    while (i < 5) {
        x ← u;
        i := i + 1;
    }
    i := 0;
    while (i < 15) {
        v ← i;
        i := i + 1;
    }
    i := 0;
    while (i < 10) {
        x ← u;
        i := i + 1;
    }
}
```

Figure 3.18: Feedback loop example

This was never a problem when L_{B} was large enough to hold all the items ever produced and consumed on the channel. In a setting where chunks need to be recycled, however, the actual top state of a monotonic buffer changes accordingly. It lies therefore not when no consumer or producer actions are possible, but just before it degenerates and stops behaving according to queue semantics.

A natural follow-up to this analysis is to propose swapping blocks whenever a monotonic buffer enters a degenerate state. While apparently very simple, it is not straightforward to implement, once we notice that there are multiple such states, to begin with: the consumer could be anywhere within the last L items (preventing the consumer from popping elements once it reaches L_{B} just reverses the problem).

This condition needs to be accounted for in the channel block-swapping code. There are basically two solutions: either producers and consumers synchronize to freeze and reflect the old consumer state into the new array (a sort of multi-word transaction), or the old array needs to be kept around to satisfy old reader requests.

3.6.3 Queues of queues

The second of the two propositions above is a lot more reasonable than the first, given how difficult—not to mention expensive—it is to manage multi-word operations. The idea of keeping multiple monotonic chunks is sound, but does not tell us how to deal with them, or more generally what our data structure should look like. We decide to take an approach based on the following remark: a queue of queues can be used loosely like a bigger queue.

If the outer (**macro**) queue has room for n little (**micro**) queues, which can contain m items each, then the whole can be operated approximately like a queue of size nm, as shown in Figure 3.19. This is only an approximation, as on the edges it temporarily allows more items to be produced than a strict nm-ring buffer would: when the producer catches up with the consumer, the outer ring can be completely filled with inner queues, with the consumer holding onto a supplementary buffer in its c field. This is precisely this trait, however, that allows consumers and producers to keep on working, without ever having to wait for each other for more than nm operations to proceed.

In the context of bounded Kahn process networks, the queue of queues is similar to a selectively bigger queue: the scheduler is only sometimes allowed to push more into it. From Lemma 1 and its proof, we conclude that it may only produce more output than the same network using regular queues. Since we are only interested in programs with sufficiently large buffers—as described in Section 3.2.3.1—in the first place, the output should actually stay identical.

Sequential queues of queues do not directly solve our concurrent block-replacement problem. However, they provide valuable insight.

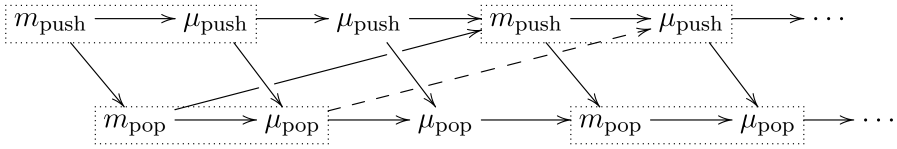
We remark that actual communication in terms of values exchanged happens only at the micro-queue level. Essentially, the queue-of-queues design allows producers and consumers to operate on multiple data-carrying smaller objects (the inner queues) as if they were a single contiguous channel. Figure 3.20 illustrates this for $n = 1$ and $m = 2$, i.e., the macro queue has a single item and each micro queue can hold two values. If we

```
struct MetaQueue {
    QueueQueue ∗qq;
    Queue ∗c;
    Queue ∗p;
};

bool push(MetaQueue ∗mq, int x)
{
    if (mq.p ≠ null) {
        if (enqueue(mq.p, x))
            return true;
    }
    var q ≔ newQueue();
    if (not enqueueQueue(mq.qq, q)) {
        freeQueue(q);
        return false;
    }
    mq.p ≔ q;
    enqueue(q, x);
    return true;
}

(bool, int) pop(MetaQueue ∗mq)
{
    if (mq.c ≠ null) {
        var ok, x ≔ dequeue(mq.c);
        if (ok)
            return true, x;
    }
    var ok, q ≔ dequeueQueue(mq.qq);
    if (not ok)
        return false;
    if (mq.c ≠ null)
        freeQueue(mq.c);
    mq.c ≔ q;
    var _, x ≔ dequeue(q);
    return true, x;
}
```

Figure 3.19: Sequential queue of queues



Macro queue of size one; micro queues of size two. Nodes are operations on the macro (m) and micro (μ) queues. Framed operations act together to simulate an overall push or pop operation. Arrows denote dependencies.

Figure 3.20: Example interaction between macro and micro operations

forget about queues of queues and concentrate only on the μ_{push} and μ_{pop} operations, then the example usage shown is exactly that of a normal single-layer queue, with push and pop matching each other (downward arrows), and the reverse dependency from each instance of pop to the next push operation that occupies the same physical location every m items (shown as a dashed arrow). Macro-queue events, denoted m_{push} and m_{pop}, add a level of indirection by periodically replacing the underlying inner queue actually used to transfer values.

Interestingly, looking at the implementation in Figure 3.19, each individual inner queue is used for exactly one round of values, undergoing m enqueue and the same number of dequeue operations before being freed. Thus, the usage pattern of micro queues precisely matches that of monotonic buffers.

From this, we can deduce, perhaps unsurprisingly, that our buffer replacement should be done following a first-in first-out queue pattern. Concretely, we need a concurrent macro queue to synchronize which blocks to use between producers and consumers. We thus conclude that to upgrade our monotonic buffer into a full-fledged concurrent queue, we need to implement a concurrent queue in the first place. Have we not gained anything from this exercise?

There are two benefits to this two-layered approach. First, if we take $m = L$ equal to the size of the actual channel we want, we only need to implement a one-item macro queue ($n = 1$), which simplifies its design. Second, as we have seen in Chapter 2, monotonic operations are cheap, while objects that allow unbounded repeating values are expensive. Therefore, it makes sense for us, from a performance standpoint, to distinguish between fast (micro) and slow (macro) operations, and keep occurrences of the second kind to a strict minimum. We thus set out to write a single-item lock-free queue to complement our monotonic buffers.

3.7 Descriptor-based macro queues

We now present a specialized macro queue object. We dub it the **register**, for it should store only one element, although we previously observed that the queue of queues sometimes temporarily behaves as if it had more, when the producer catches up with the consumer.

3.7.1 Process-controlled channels

To integrate with Kahn processes, which may be handled by multiple worker threads at the same time, our register is built around the classic concept of descriptors, previously introduced in Section 2.8.1. A descriptor is a structure that represents a macro-level operation on the channel: macro push or macro pop. Even though both specify an operation to execute, unlike a simple method call, a descriptor exposes a visible shared state that can be read and updated by threads. At any time, it is either **complete**, if the associated operation has been performed successfully, or incomplete, if it is still pending. Fundamentally, we want to guarantee that multiple applications of a same descriptor are idempotent, so assistance is straightforward: simply apply the current descriptor until it is complete. This is similar to the role played by the *status* field in descriptors for the two-word swap discussed in Sections 2.8.1 to 2.8.3.

As we have seen in Section 2.8.1, the appearance of descriptors is linked to disjoint-access multi-word operations. In the case of channel descriptors, this is caused by the need for checkpoints, as defined in Section 3.6.1. We recall that process states need to be saved along with channel block swaps, so that no further computation depends on the data contained in the old chunk we are throwing away.

We want to update in a single transaction both one side (producer or consumer) of a channel and the attached process. Furthermore, this operation may span many different, disjoint, process–channel couples, thus we opt for the more complex multi-word update instead of a simpler design where the two would be fused in a same block. If we did that, by transitivity, we would end up representing whole connected components in the Kahn process graph as single chunks!

In place of a full-fledged double compare-and-swap mechanism, as we presented in Section 2.8.3, here, we propose a cheaper solution, by taking advantage of some properties of our process networks:

- Instead of allocating a separate descriptor object, we bundle descriptors with process states. Each chunk contains thus both a valid process state and a descriptor. Its effective value depends on the status of the descriptor part. If it is incomplete, then the block is taken to represent the descriptor; if it is complete, then the transaction is finished and the actual process state may be accessed normally.

- In the two-pointer swap example of Section 2.8.3, we first overwrite both variables affected by the transaction with the descriptor. Here, although we are also updating two locations (one in the process structure and another in the channel), we only need to set the descriptor on the process side. Overwriting locations with the descriptor only serves to stabilize the shared state, so that no renegade thread makes a divergent modification to one of the variables involved in the transaction while it is underway. However, in our case, we only ever replace buffers that are maximal on the side we are working on: we only push new blocks when the previous ones have no space left, and only pop old blocks when all the values have been read. Therefore, no conflicting concurrent change can occur, because there is no other possible

```
struct Desc;

struct State {
    ...
    int chi;
    Desc descs[];
};

struct Proc {
    ...
    State * atomic state;
    ...
};
```

As a syntactic note, fields marked *atomic* may be both read and written concurrently by several threads at a same time. Other variables are at most accessed read-only by multiple threads, and may be written to in non-concurrent code, e.g., during thread-local initialization, before the object is made visible through the shared data structure.

Figure 3.21: Channel–process integration

action to take once the block is maximized. Further stabilizing is unnecessary; we only need to protect against different threads assisting a same transaction.

Put together, this provides us with a skeleton for the structure of processes and states, which we sketch in Figure 3.21. Each process is represented by a fixed shared *Proc* object, spawned during reconfiguration. Worker threads share process states through *State* blocks. Since processes may be (and usually are) bound to multiple channels, we allow multiple descriptors to coexist within a single *State* chunk, as part of the *descs* array. Conversely, a channel is logically associated with exactly two sequences of descriptors, on each side, stored in *State* objects. Macro operations occur as mandated by those descriptors. The *stateOfDesc* function, Figure 3.22, returns the *State* object containing a given descriptor.

Process states may not be updated freely, though. To guarantee the proper operations of the channels, we must make sure to follow two simple rules, which ensure that assumptions we have previously made hold true:

Client alternation Although we store descriptors along with process states, they still virtually represent a value distinct from the state itself. Descriptors need assistance and any thread that reads an incomplete descriptor must help finish its transaction, without exception. This rule forces modifications of the process state (i.e., the *state* field) to strictly alternate with complete macro operations. They happen one after another in lockstep.

```
// Returns the State instance that contains the given descriptor.
// This function is only used in assertions.
State ∗stateOfDesc(Desc ∗d, int chi)
{
    // Not strictly valid C, but this is how offsetof is usually implemented.
    var offset := (char ∗)&((State ∗)0).descs[chi] − (char ∗)0;
    return (State ∗)((char ∗)d − offset);
}
```

Figure 3.22: Process state from descriptor

Relevant descriptors We assume that each process state change may introduce at most one new descriptor, replacing a previous complete one at the same position in the *descs* array. Others must be copied over unaltered. The *chi* field specifies which descriptor is relevant for a specific state. This rule is a natural consequence of the nature of Kahn processes: at any checkpoint, a process may be waiting on at most one channel. In practice, in the rest of this section, we only consider relevant descriptor updates, and generally disregard copies.

Intuitively, we can say that this overall construction enforces the process-level single-producer single-consumer constraint. In essence, the channel methods themselves only handle strictly sequential requests exposed through descriptors. As is standard in lock-free fare, we have transformed the blocking problem of exclusive single-producer or single-consumer channel operations (at the Kahn process level) into a lock-free word-sized election problem (on *state* fields), where the winner dictates the next (macro) operation, which may then be assisted by other threads.

3.7.2 The channel structure

The *Chan* type of Figure 3.23 represents a channel, that is, in our case, a one-element queue of monotonic blocks used as a larger queue. It should thus serve the same function as the *MetaQueue* structure from Figure 3.19. However, they look quite different.

The first thing we notice is that *Chan* contains only two fields. Both fields taken together essentially describe just one array: *data* has two slots for *Mono* pointers, each one referring to a micro queue of *len* items. Of particular interest is the fact that there does not seem to be any equivalent of the *c* and *p* fields of *MetaQueue*.

The reason for this has to do with how memory chunks are handled in a lock-free system. Usually, it is best to refrain from moving block references around as much as possible. The reason is that moving things requires at least a two-word operation that covers the source and destination, which we would rather avoid. We could copy the pointers, but it blurs questions of ownership—who gets to call *destroy* on the block.

Instead, we choose to add one layer of indirection, as follows. In place of moving micro queues from the macro queue to the client-specific variables, we let the producers

```
struct Desc {
    Chan *ch;
    unsigned int turn;
    Mono *block;
};

struct Mono {
    unsigned int turn;
    atomic bool empty;
    atomic int buf[];
};

struct Chan {
    int len;
    Mono * atomic data[2];
};

void makePush(Chan *, Desc *, Desc *);
void makePop(Chan *, Desc *, Desc *);

bool macroPush(Desc *, Proc *, int);
bool macroPop(Desc *, Proc *, int);
Mono *getMono(Desc *d, Proc *, int);
```

Again, only fields marked *atomic* may be both read and written concurrently. Therefore, *len* does not change after initialization, while we can expect *data* to be modified further by the algorithm.

Figure 3.23: Descriptor-based channel interface

and consumers access the last monotonic buffer pushed (by *macroPush*) or popped (by *macroPop*), respectively, directly in the *data* array. In order to do so, all operations—hence descriptors—must be indexed: the **turn** (held in a field of the same name) is the number of previously completed operations on the same side. The first descriptor has turn zero, and each subsequent one increments the turn by one.

Macro push and pop invocations add and remove elements only at their given turn, accessing the *data* array in round-robin fashion, such that all even turns get mapped to index zero, and all odd turns to index one. The producer side is responsible for swapping in new blocks, while consumers must mark removable *Mono* blocks as such by setting their *empty* bit. The combination of both descriptors and the *empty* state of the current blocks determines the state of the register, as shown in Figure 3.24.

This diagram demonstrates all possible (legal) states of the shared variables making up the register data structure, starting with 0_{PC} after being initialized according to Figure 3.25. We do not represent almost constant fields such as *turn*, which do not change after initialization (before they get plugged into the shared object). Of particular note, only states that do not feature an underlined marker are stable; others are transient, as they involve incomplete descriptors.

As we can see from the state diagram, each transaction related to a macro operation consists of two parts:

1. The descriptor is replaced, as the result of swapping in a new *State* object. New descriptors are created by the *makePush* and *makePop*, in Figure 3.26. This corresponds to the arrow from an unmarked letter to the same underlined letter. Which letter changes depends on which descriptor is refreshed.

2. The descriptor is completed, as the result of a call to *macroPush* or *macroPop*. This corresponds to the arrow from an underlined letter to the same unmarked letter. Again, which letter changes depends on which method completes.

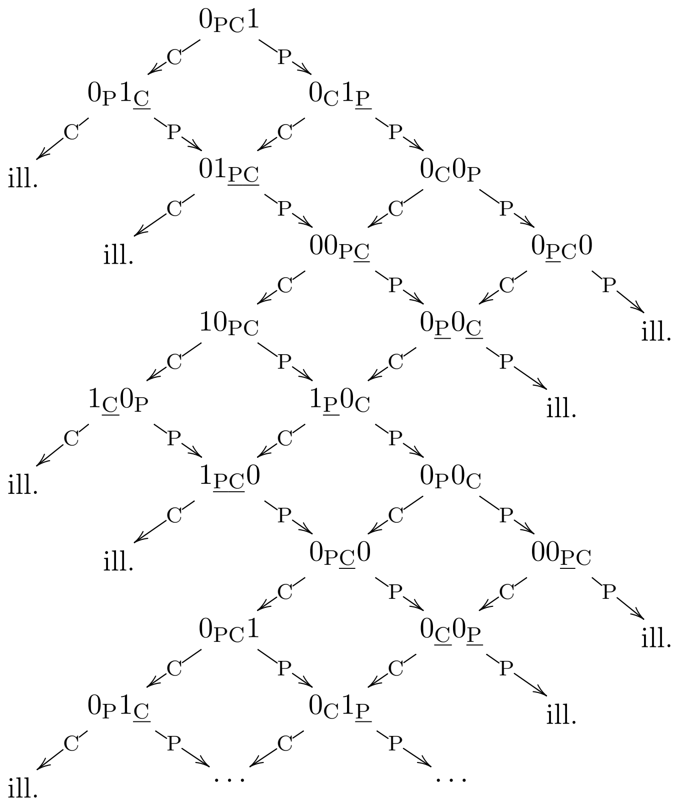
In a concurrent execution, many more things can happen. However, we will prove that the range of possible behaviors is limited to what is shown in Figure 3.24. The succession of stable states in a concurrent history can then be taken as a linearization of the corresponding execution.

3.7.3 An example of macro-queue usage

To conclude these preliminary discussions of the macro queue, and before diving into the concurrent code proper, let us examine a step-by-step example of process–channel interaction. Figure 3.27 illustrates how the various functions making up the macro-queue interface (as defined in Figure 3.23 and described above) might be called together by some worker thread, in order to perform their channel duties.

3.7.4 Pushing into the macro queue

We first study the code for the macro push, given in Figure 3.28. The arguments are as follows:



The two digits represent the *data* array, where each digit is the value of the *empty* flag in the corresponding block. The P and C subscripts indicate the turn associated with the current producer and consumer descriptors, respectively. They are underlined if the operation is incomplete. *Ill.* denotes an illegal state. For example, $0_C1_{\underline{P}}$ is the state in which the first *data* element is not empty, the second is, the producer descriptor is incomplete and has an odd turn value, while the consumer descriptor is complete and has an even turn value.

Figure 3.24: Register states

```
void makeChan(Chan *ch, int len)
{
   ch.len := len;

   Mono *a := malloc(sizeof *a + len * sizeof *a.buf);
   memset(a.buf, 0, len * sizeof *a.buf);
   a.turn := 0;
   a.empty := false;
   ch.data[0] := a;

   Mono *b := malloc(sizeof *b + len * sizeof *b.buf);
   memset(b.buf, 0, len * sizeof *b.buf);
   b.turn := −1;
   b.empty := true;
   ch.data[1] := b;
}
```

```
// Copies prevd to d for process initialization purposes during reconfiguration;
// prevd should be non-null for inherited channels and null for new channels.
void copyPush(Chan *ch, Desc *d, Desc *prevd)
{
   d.ch := ch;
   d.turn := prevd = null ? 0 : prevd.turn;
   d.block := prevd = null ? ch.data[0] : prevd.block;
}
```

```
// Same as copyPush, but for consumers.
void copyPop(Chan *ch, Desc *d, Desc *prevd)
{
   d.ch := ch;
   d.turn := prevd = null ? 0 : prevd.turn;
   d.block := null;
}
```

Figure 3.25: Descriptor-based register: initialization

```
void makePush(Chan *ch, Desc *d, Desc *prevd)
{
    d.ch := ch;
    d.turn := prevd.turn + 1;
    d.block := malloc(sizeof *d.block + ch.len * sizeof *d.buf);
    d.block.turn := d.turn;
    d.block.empty := false;
    memset(d.block.buf, 0, len * sizeof *d.buf);
}
```

```
void makePop(Chan *ch, Desc *d, Desc *prevd)
{
    d.ch := ch;
    d.turn := prevd.turn + 1;
    d.block := null;
}
```

Figure 3.26: Descriptor-based register: *make* methods

```
Proc *p := ...;
var s := safeRead(&p.state);
Chan *ch := ...;

var t := cloneState(s);
makePush(ch, &t.descs[0], &s.descs[0]);

beginReadSection(t);
cas(&p.state, s, t);
macroPush(&t.descs[0], p, 0);

var a := getMono();
var buf := (int[*])(ch.len * 3, a.buf)
monoPush(buf, 0, 0xDEADBEEF);
```

A macro push followed by a micro push on the new buffer. States are copied using a fictional *cloneState* function which is assumed to exist. The code assumes all fallible calls (e.g., *cas*, *macroPush*, *getMono*) are successful.

Figure 3.27: Example macro-queue usage

```
 1 bool macroPush(Desc *d, Proc *p, int chi)
 2 {
 3    assert(inReadSection(stateOfDesc(d, chi)));
 4    var a := safeRead(&d.ch.data[d.turn % 2]);
 5    bool status;
 6    var empty := a.empty;
 7    if (&p.state.descs[chi] $\neq$ d)
 8       status := true;
 9    else if (not empty)
10       status := false;
11    else {
12       if (a $\neq$ d.block) {
13          if (cas(&d.ch.data[d.turn % 2], a, d.block))
14             destroy(a, free);
15       }
16       status := true;
17    }
18    endReadSection(a);
19    return status;
20 }
```

Figure 3.28: Descriptor-based register: *macroPush*

d a protected pointer to the descriptor of the push operation to execute, which must belong to a *State* block safe-read by the caller;

p the process to which the descriptor is bound;

chi the index of the parameter of *p* (as a Kahn process) to which the channel is bound.

The *macroPush* method performs the requested operation and returns true if it has been completed, or false if it cannot yet be fulfilled. For a macro push operation, completion is defined as successfully executing line 13 (no other line writes to shared memory), at which point the register state changes: the corresponding index in the *data* array turns from an old empty block to a new non-empty one, which can be consumed by further macro pop calls.

The life of a register is punctuated by two kinds of events of particular importance to the producer side: producer descriptor swaps, and *data* block changes. The responsibility for these modifications falls entirely upon producers. Provided the caller follows strict client alternation as specified above, let us show that the alternation property extends to *data* block changes.

Lemma 6. *If an invocation of* macroPush *returns true, it happens after a compare-and-swap on line 13 executed with the same descriptor (either in the same invocation or another).*

Proof. There are only two paths that return true in *macroPush.* The first one requires the test on line 7 to branch out, which implies the descriptor passed to the method has been swapped out. By the client alternation protocol, this only happens after a previous true return from another instance of *macroPush* with the same descriptor. Thus, the first such occurrence must pass through line 13. □

Given a channel and history, let $\alpha_0 \dots \alpha_{a-1}$ be the sequence of changes to the *data* field (at either index), and $\pi_0 \dots \pi_{p-1}$ the sequence of relevant producer descriptor changes. (Index zero denotes initialization, counted as a modification.) Per convention, when there is no possible confusion, we also use those notations to refer to the values thus instated, in addition to the modifications themselves, as memory events.

Lemma 7. *The sequences α and π alternate such that $p = a \vee p = a + 1$ and $\forall k$, $\pi_k \prec \alpha_k \prec \pi_{k+1}$, where $\prec$ denotes the happens-before relation, and each α_k swaps in the block described by π_k.*

Proof. As usual, we suppose that only our register code (hence only *macroPush*) changes the *data* field. We proceed by induction on the number of descriptor changes in a history.

Initially, the property is verified since initialization occurs sequentially and the initial descriptor π_0 and blocks match, therefore no compare-and-swap occurs at all, and all invocations of *macroPush* fail on line 12.

Suppose the property holds for all histories such that $p \leq n$. Let us show it for $n+1$. Suppose we have a history containing $p = n + 1$ producer descriptor changes.

Consider a call A to *macroPush* whose successful modification of the *data* field occurs after π_n. Let π_k be the descriptor passed to A as its third argument. Let us show that $k = n$.

Suppose $k < n$. Because of client alternation, π_{n-1} must be complete, thus, by Lemma 6, the history must contain α_{n-1}. By induction hypothesis, history looks like the following around π_k. (Since $k < n$, this structure is maintained until at least π_{k+1}.)

$$\alpha_{k-2} \rightarrow \pi_{k-1} \rightarrow \alpha_{k-1} \rightarrow \boxed{\pi_k} \rightarrow \alpha_k \rightarrow \pi_{k+1} \rightarrow (\alpha_{k+1} \rightarrow \pi_{k+2} \rightarrow \alpha_{k+2})$$

The value of π_k is assumed to be safe-read by the caller, therefore a successful double check on line 7 (that does not branch out) must happen before π_{k+1}. Thus, line 4 must read either α_{k-2} (the previous value at the same parity) or α_k, depending on whether it happens before or after α_k.

- Suppose we load α_{k-2}. Since line 4 performs a safe read and only fresh pointers are swapped into *data*, α_{k-2} stays unique during A, and thus cannot be overwritten for compare-and-swap to succeed. Therefore, line 13 must happen before α_k; thus $\pi_n \prec \alpha_k$, which is impossible since $k < n$.

- Suppose we load α_k on line 4, then the same-block test on line 12 does not branch into the compare-and-swap. Impossible.

```
 1 bool macroPop(Desc ∗d, Proc ∗p, int chi)
 2 {
 3    assert(inReadSection(stateOfDesc(d)));
 4    var a := safeRead(&d.ch.data[d.turn % 2]);
 5    var b := safeRead(&d.ch.data[(d.turn − 1) % 2]);
 6    bool status;
 7    var empty := a.empty;
 8    if (&p.state.descs[chi] ≠ d)
 9       status := true;
10    else if (empty)
11       status := false;
12    else if (b.turn ≠ d.turn − 1)
13       status := true;
14    else {
15       b.empty := true;
16       status := true;
17    }
18    endReadSection(b);
19    endReadSection(a);
20    return status;
21 }
```

Figure 3.29: Descriptor-based register: *macroPop*

Therefore, $k = n$.

Let us consider all the invocations of *macroPush* that successfully modify *data* during the stint of π_n as current descriptor. They must all receive π_n as descriptor, as per the above. Furthermore, all such calls compete to set *data* to the same block pointer prescribed by π_n. Since that block is freshly fetched from the global allocator, it is different from α_{n-2}, which is live when π_n is first instated. Therefore, exactly one change occurs: the first to swap from α_{n-2} to α_n, after which subsequent calls either exit on line 12, or fail during compare-and-swap because of their outdated expectations.

We have thus verified the property for $n + 1$. □

3.7.5 Popping from the macro queue

We now look at the consumer side, in Figure 3.29. A *macroPop* invocation completes its descriptor upon writing to the *empty* field on line 15.

We extend the above notations to include $\kappa_0 \ldots \kappa_{c-1}$ as the sequence of relevant consumer descriptor changes, and $\beta_0 \ldots \beta_{b-1}$ as the sequence of *empty* field changes on a live block (linked from the register shared structure or held in a read section).

Lemma 8. *The sequences α, β and κ alternate such that:*

1. *$b \leq c \leq b+1$ and $b \leq a \leq b+1$, hence $a = c \pm 1$;*
2. *for all k, $\kappa_k \prec \beta_k \prec \kappa_{k+1}$;*
3. *each β_k sets the empty flag of the corresponding block α_{k-1} when it is current, and $\alpha_k \prec \beta_k \prec \alpha_{k+1}$;*
4. *and every true return from* macroPop *with descriptor κ_k happens after the matching β_k.*

Proof. We safe-read both blocks on lines 4 and 5 and expect that no other code touches those memory locations while they are in use in our register. Thus, only *macroPop* flips the *empty* field from false to true on a block while it lives in a register.

We proceed by induction on the number of descriptor changes in a history. Initially, the property is verified since initialization occurs sequentially and the block at index -1 is initialized to empty, hence every *macroPop* call on the initial descriptor κ_0 fails on line 7.

Suppose the property holds for all histories such that $c \leq n$. Let us show it for $n+1$. Suppose we have a history containing $c = n+1$ consumer descriptor changes.

Because of client alternation, κ_{n-1} must be complete. Therefore, the history must contain a true-returning invocation of *macroPop*, which, by the fourth point of the induction hypothesis, happens after β_{n-1}. Therefore:

$$\beta_{n-1} \prec \kappa_n \tag{3.1}$$

Let us show first show the two first points of the induction hypothesis for $n+1$:

$$b \leq c \leq b+1 \wedge b \leq a \leq b+1 \tag{3.2}$$

$$\forall k,\ \kappa_k \prec \beta_k \prec \kappa_{k+1} \tag{3.3}$$

For any $k < n$, the induction hypothesis tells us that history looks like the following around κ_k. The dotted frame indicates those events thare are not strictly ordered with regard to κ_k. (Since $k < n$, this structure is maintained until at least κ_{k+1}.)

$$\begin{tikzcd}
\alpha_{k-2} \arrow[r] \arrow[dr] & \pi_{k-1} \arrow[r] & \alpha_{k-1} \arrow[r] \arrow[dr] & \pi_k \arrow[r] & \alpha_k \arrow[r] \arrow[dr] & (\pi_{k+1} \arrow[r] & \alpha_{k+1} \arrow[r] \arrow[dr] & \pi_{k+2} \arrow[r] & \alpha_{k+2}) \\
 & \beta_{k-2} \arrow[r] \arrow[ur] & \kappa_{k-1} \arrow[r] & \beta_{k-1} \arrow[r] \arrow[ur] & \boxed{\kappa_k} \arrow[r] & \beta_k \arrow[r] \arrow[ur] & \kappa_{k+1} \arrow[r] & (\beta_{k+1} \arrow[r] \arrow[ur] & \kappa_{k+2})
\end{tikzcd}$$

- Consider a call B to *macroPop* whose successful store (that replaces false with true) into the *empty* field on line 15 occurs after κ_n. Let κ_k be the descriptor passed to B as its third argument. Let us show that:

  $$\kappa_n \prec \beta_{k'} \wedge [\beta_{k'} \text{ has descriptor } \kappa_k] \implies k = n \tag{3.4}$$

  Suppose $k < n$. Similarly to the producer side, the value of κ_k is assumed to be safe-read by the caller, therefore a successful double check on line 8 (that does not branch out) must happen before κ_{k+1}. Thus, line 5 must read either α_{k-1} or α_{k+1}, depending on whether it happens before or after α_{k+1}.

By the first point of our induction hypothesis, they cannot load any newer value, by virtue of the read statements taking place in a prefix history that stops right before κ_{k+1}, hence with only $c = k + 1$.

- If it is α_{k-1}, then by the time B reaches line 15, after κ_n, hence after κ_{k+1}, β_k will have already occured on the same block (by the third point of our induction hypothesis). Therefore B does not change the *empty* field.
- If it is α_{k+1}, then the turn test on line 12 fails, assuming the *turn* field can hold at least three different values, so that for any turn value t, $t-1$ and $t+1$ are different. Thus B does not reach line 15.

- Consider a call A to *macroPush* whose successful modification of the *data* field occurs after κ_n. Let π_k be the descriptor passed to A as its third argument. Let us show that:

$$\kappa_n \prec \alpha_k \implies n \leq k \leq n+1 \tag{3.5}$$

By Lemma 7, A must contain α_k. If $k + 1 \leq n - 1$, then by induction hypothesis on the shorter history that stops at κ_{k+1}, $\alpha_k \prec \kappa_{k+1} \prec \kappa_n$, which is impossible. Therefore, $k \geq n - 1$.

Yet, if $k = n-1$. We recall that $\beta_{n-1} \prec \kappa_n$ by Equation (3.1). Therefore, induction hypothesis again yields $\alpha_{n-1} \prec \beta_{n-1} \prec \kappa_n$. Impossible.

If $k > n+1$, then α_{n+2} must exist in the history. By induction hypothesis, it cannot happen before κ_n. By Lemma 7, α_{n+2} takes π_{n+2} as argument, hence its turn is $n+2$. It needs not to branch out because of line 6: the *empty* bit must be set on the block it replaces, which is π_n. Since only *macroPop* sets the *empty* flag, there must be a $\beta_{k'}$ that matches α_n. However, by induction hypothesis, any $\beta_{k'}$ happening before κ_n must have $k' \leq n - 1$ and is already matched with the corresponding $\alpha_{k'-1}$. Conversely, we have shown as Equation (3.4) that any $\beta_{k'}$ that happens after κ_n takes κ_n as argument, and therefore target the *empty* flag of a block with the same parity as $n - 1$. Such a block cannot be α_n. In either case, none of them can match α_n. Therefore, $k \leq n + 1$.

Let us review the different clauses we need to prove. We have just proven the new bounds on a, b and c for $n+1$. Also, for all β_k, we have shown that $k < n \implies \beta_k \prec \kappa_n$. Points 3 and 4 remain.

Let us now show that:

$$\forall k,\ [\beta_k \text{ empties block } \alpha_{k-1}] \wedge \alpha_k \prec \beta_k \prec \alpha_{k+1} \tag{3.6}$$

$$\forall k,\ \kappa_n \prec \beta_k \implies k = n \tag{3.7}$$

- Let us now consider all the calls B to *macroPop* that qualify for β status, i.e. they change *empty* from false to true, after κ_n. By Equation (3.4) above, they all receive κ_n as descriptor. On one side, Equation (3.5) tells us that the history contains no α_k greater than α_{n+1}. On the other side, we know by Equation (3.1) that $\alpha_{n-1} \prec \beta_{n-1} \prec \kappa_n \prec \text{call}(B)$.

  Thus, B loads either α_{n-1} or α_{n+1} on line 5, and by the same reasoning as we used to derive Equation (3.4), it must load α_{n-1}, else the turn check on line 12 would fail. Therefore, if it indeed mutates *empty* from false to true, it can only do so on α_{n-1}. This also implies that there is at most one such call, since only the first can change the bit on α_{n-1} from false to true.

- Conversely, Equation (3.5) tells us that only α_n or α_{n+1} may happen after κ_n.

  - α_n either happens before κ_n or after. If it happens before, then the induction hypothesis applies and $\beta_{n-1} \prec \alpha_n \prec \kappa_n \prec \beta_n$ (if any β_n exists).

    If it happens after, then by Equation (3.1) $\beta_{n-1} \prec \kappa_n \prec \alpha_n$. If β_n exists, it must happen after α_n. Indeed, in the *macroPop* invocation associated with β_n, line 4 follows κ_n, hence α_{n-2}, and there is no α_{n+2} or higher in the current history. Thus it loads either α_n or α_{n-2}, depending on whether it happens before or after α_n. However, since it happens after κ_k, the *empty* bit on α_{n-2} is already set by $\beta_{n-1} \prec \kappa_n$, making the test on line 7 fail; and since n is maximal in the current history, line 8 does not override it. Thus, it can only be α_n, and we have $\beta_{n-1} \prec \alpha_n \prec \beta_n$.
  - α_{n+1} requires the *empty* bit to be set on α_{n-1}. Following a similar reasoning to what we did for Equation (3.5) above, this condition cannot be satisfied by a previous $\beta_{k'}$ with $k' < n$, which is matched with $\alpha_{k'-1}$ by induction hypothesis. Therefore, it must be the only invocation of *macroPop* (as we have seen above) with κ_n that flips the bit on α_{n-1}. Thus, $\beta_n \prec \alpha_{n+1}$.

We are left with proving that return values from *macroPop* are consistent for all invocations that return after κ_n. Let B be such a call, with descriptor κ_k, let us show that:

$$\forall B,\ [B \text{ takes } \kappa_k] \wedge [B \text{ returns true}] \implies \beta_k \prec \text{return}(B) \tag{3.8}$$

If the return statement happens after κ_n but the test that branches to it happens before, then the method could have returned already in a shorter history that excludes κ_n. Thus, by induction hypothesis, $\beta_k \prec \text{return}(B)$. Suppose the test itself happens after κ_n.

- If B returns after line 8, then $k \leq n-1$, and by induction hypothesis and Equation (3.1), β_k happens before it.

- If B returns after line 12, then by the double check on line 8, it must be that $n-1 \leq k \leq n$. By Equations (3.3) and (3.6), and due to parity, failing the test means B loads α_{k+1}, which happens after β_k by Equation (3.6) again.

- Otherwise, B contains β_k.

We have thus shown all the points of the induction property for $n+1$, and the induction finally holds for any n. □

Corollary 3. *If an invocation B of* macroPop *with descriptor κ_k returns false, then it must have been called between κ_k and α_k.*

Proof. The call to B happens before κ_{k+1}. From Lemma 8, it therefore happens after α_{k-1} and before both α_{k+2} and β_{k+1}. It can therefore read only either α_{k-2} or α_k on line 4, due to parity. B returns false, but the bit on α_k is set by β_{k+1} (Lemma 8), which happens after the double check, line 8, thus after checking for *empty* on line 7.[6] The latter must thus check against α_{k-2}. Therefore, the call must occur before α_k, which replaces α_{k-2}. □

Corollary 4. *If an invocation A of* macroPush *with descriptor π_k returns false, then it must have been called between π_k and β_{k-1}.*

Proof. Symmetrically, from the proof of Corollary 3. □

3.7.6 A linearizable macro queue

We can now put the above pieces together into a linearizable macro queue.

Lemma 9. *Register methods* macroPush *and* macroPop *are linearizable.*

Proof. Lemmas 6 to 8 and Corollaries 3 and 4 give us a straightforward set of linearization points. Invocations of *macroPush* and *macroPop* that contain α or β events linearize at those points. Other instances—which do not affect the shared state—linearize at their return point if they return true, or when they are called if they return false.[7]

Below is a subset of the state diagram of Figure 3.24, annotated with the memory events—that double as linearization points—we have defined.

[6] This explains why the value of *empty* is loaded before the double check, but the branch occurs after. This is to ensure the *empty* bit read is consistent with the array matching the descriptor; e.g., we want to make sure that calls to *macroPop* that begin after β_k do not get delayed enough that they mistake the *empty* bit of the next same-parity chunk for their own.

[7] Of course, we could have taken other linearization points for those invocations, but this option is the simplest, since calls on a same descriptor are supposed to monotonically go from “cannot be fulfilled” to “state changing” to “too late, already over,” so it makes sense for the groups on each side to pick their linearization points as far as possible.

$$\begin{tikzcd}
 & 0_{\mathrm{PC}}1 \arrow[dl, "\kappa"'] \arrow[dr, "\pi"] & & & \\
0_{\mathrm{P}}1_{\underline{\mathrm{C}}} \arrow[dr, "\pi"] & & 0_{\mathrm{C}}1_{\underline{\mathrm{P}}} \arrow[dl, "\kappa"'] \arrow[dr, "\alpha"] & & \\
 & 01_{\underline{\mathrm{PC}}} \arrow[dr, "\alpha"] & & 0_{\mathrm{C}}0_{\mathrm{P}} \arrow[dl, "\kappa"'] \arrow[dr, "\pi"] & \\
 & & 00_{\mathrm{P}\underline{\mathrm{C}}} \arrow[dl, "\beta"'] \arrow[dr, "\pi"] & & 0_{\underline{\mathrm{P}}\mathrm{C}}0 \arrow[dl, "\kappa"'] \\
 & 10_{\mathrm{PC}} \arrow[dl, "\kappa"'] \arrow[dr, "\pi"] & & 0_{\underline{\mathrm{P}}}0_{\underline{\mathrm{C}}} \arrow[dl, "\beta"'] & \\
1_{\underline{\mathrm{C}}}0_{\mathrm{P}} \arrow[dr, "\pi"] & & 1_{\underline{\mathrm{P}}}0_{\mathrm{C}} \arrow[dl, "\kappa"'] \arrow[dr, "\alpha"] & & \\
 & \cdots & & \cdots &
\end{tikzcd}$$

The alternation of stable and unstable states is guaranteed by the alternation properties of α and π on the producer side (Lemma 7), and β and κ on the consumer side (Lemma 8). Guarding against illegal states is a result of the relation between the bounds a, b, c and p (Lemmas 7 and 8), which prevents any unwanted accumulation on a single side.

Therefore, invocations of *macroPush* and *macroPop* that modify the shared state of the register linearize into the corresponding abstract state changes.

Futhermore, Lemmas 6 and 8 tell us that other invocations that return true do so as if they atomically happened at any point after the corresponding state change has been effected (and would thus also return true in a sequential context).

Conversely, from Corollaries 3 and 4, we can say that false-returning invocations behave as if executed on the spot, when they are called.

Finally, every invocation of *macroPush* or *macroPop*, whether it returns true or false, whether it affects the shared state or not, is linearizable. □

3.7.7 Data block access

Now that we have a linearizable queue of blocks, we still need a way to access those blocks, in order to carry out the monotonic operations that actually matter. This is accomplished by calling the *getMono* function, shown in Figure 3.30, on a complete descriptor.

The semantics of the *getMono* function is very simple: if called with a current descriptor, it returns the associated data block; otherwise, it returns null. It is linearizable as a direct consequence of the alternation structure of descriptors and data arrays proven above.

Lemma 10. *The* getMono *method is linearizable.*

Proof. It linearizes when testing whether the descriptor is still current, on line 5.

In the producer case, suppose we receive π_k as argument. By client hypothesis, the call happens after α_k completes π_k. Then, either line 5 happens before π_{k+1} or it does

```
 1  Mono *getMono(Desc *d, Proc *p, int chi)
 2  {
 3     assert(inReadSection(stateOfDesc(d)));
 4     var a := safeRead(&d.ch.data[d.turn % 2]);
 5     if (&p.state.descs[chi] ≠ d) {
 6        endReadSection(a);
 7        a := null;
 8     }
 9     return a;
10  }
```

Figure 3.30: Descriptor-based register: *getMono*

not. If it does, then by Lemma 7, there is a single α_k that can be read on line 4, which is the same as if we ran the function sequentially at that point. If it does not, then we return null.

In the consumer case, suppose we receive κ_k as argument. By client hypothesis, the call happens after β_k completes κ_k, hence after α_k. Then, either line 5 happens before or after κ_{k+1}. If before, then by Lemma 8, there is only one corresponding α_k with the same parity to be read on line 4. If after, then we return null.

In all cases, a sequential invocation at the point of the test would have the exact same effects. □

3.8 Semi-local layer

We finally get to build an entire process network from all the pieces we have gathered along the way. The goal is to add the buffer-reuse capabilities of Section 3.7 to the initial design developed in Section 3.6. As we have seen, maintaining entirely local copies of process states is not satisfactory. However, it helps keep contention low. We therefore proceed by adding process sharing between worker threads as needed: whenever recycling is in order.

Our new graph data object has a two-level structure: the semi-local (or monotonic) level, and the shared level. Normal operations execute at the semi-local level, until a major event that must be synchronized (e.g., a buffer being replaced) happens, at which point, a more expensive consensus-based synchronization phase plays out.

The overall structure of the interpreter is outlined in Figure 3.31. The nodes in the diagram represent data types. We can see the process graph on the left-hand side, made of an alternation of *Proc* (process), *Chan* (channel) and *Conf* (configuration) objects. We introduced the concurrent *Proc* and *Chan* types in Section 3.7.1, and the *Conf* structure is the same as the sequential version, defined in Section 3.2.4. The three kinds of arrows show different sharing relations:

- A plain arrow (e.g., between *Proc* and *Chan*) indicates that one object has a con-

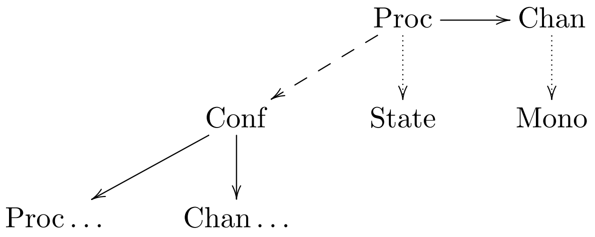


Figure 3.31: Memory layout of the concurrent implementation

stant (after initialization) field pointing to the other. Those links are set before the objects are made visible in the shared structure.

- A dashed arrow (e.g., between *Proc* and *Conf*) denotes a monotonic pointer that may go from null to point to an instance of the described structure. In the case of *Proc* and *Conf*, a process starts with no children (no linked *Conf* object) and may be reconfigured once, by swapping a pointer to point to the new configuration.
- Finally, a dotted arrow (e.g., between *Proc* and *State*) represents the most permissive relation, where the pointer from one structure to the other may be updated multiple times, using compare-and-swap, according to the various protocols described in this chapter (e.g., macro operations to change *Mono* buffers).

As a rule of thumb, operations that update links in the shared process graph structure (the non-plain arrows in Figure 3.31) are rather expensive, as they involve compare-and-swap.

Macro operations, presented in Section 3.7, are more expensive than the simple micro operations of Section 3.6. Writing to or reading from a *Mono* queue chunk is very cheap, as it requires only load and store instructions.

The same applies to process state updates. Replacing the *State* object pointed to by a process is a costly endeavor. However, as explained in Section 3.6, we do not need to keep process states perfectly synchronized between threads at all times. The only times when we absolutely must update the state pointer is when setting up a new descriptor for a macro operation, according to the protocol detailed in Sections 3.7.1 to 3.7.3. The entire purpose of the semi-local layer is thus to enable worker threads to locally interpolate process states for use in conjunction with monotonic channel buffers, such that, eventually, an average Kahn process transition takes zero read-modify-write operations.

3.8.1 Shared process graph

We start with a look at the shared data structure, in Figure 3.32. The biggest change compared to Figure 3.10 is the change from an integer array to a dedicated structure for the *state* field. The *State* structure contains the following members:

final True if this is a final state on which *step* should not be called anymore.

```
struct State {
    bool final;
    int *vars;
    int *inds;
    atomic int chi;
    Desc descs[];
};

struct Proc {
    bool step(Arg[*], int[*]);
    Arg args[*];
    int nvar;
    State * atomic state;
    Conf *reconfigure(Arg[*], int[*]);
    Conf * atomic conf;
    atomic int njoined;
    Proc *parent;
};
```

Figure 3.32: Kahn process graph data structure

vars Process variables, the contents of which is the same as the old *state* field.

inds Channel indices, for micro (monotonic) operations, with one entry for each connected channel, mirroring the order of the *args* array.

chi Index of the descriptor describing the ongoing macro operation; monotonically falls down to -1 to signify completion of the most recent macro operation.

descs Descriptors for macro operations; again, one entry per channel argument.

Regarding shared *Proc* nodes, we manage memory using a bin, as described in Section 3.4.1. Since we do not support multiple reconfigurations per process, the network strictly grows until all processes stop as there is no notion of graph contraction symmetric to reconfiguration. We can apply the tree bin technique to the entire process tree. The *Proc* objects make up the nodes of the tree, with children linked by the *conf* field of their parent. In fact, since reconfiguration allocates arrays of *Proc* objects at a time, tree bin nodes are really arrays of processes rather than single objects.

3.8.2 Local process graph

For our strategy to work, each worker thread must also have its own shadow copy of process states, at the very least, and may hold onto a few more things to speed up the computation, or just make our life easier. Essentially, it acts as a local **cache**, that is, a

```
typedef Cache[‘K, ‘V];

‘K ∗getCached(Cache[‘K, ‘V] ∗, ‘K ∗);
void putCached(Cache[‘K, ‘V] ∗, ‘K ∗, ‘V ∗, void (∗)(‘K ∗, ‘V ∗));

thread_local Cache[Proc, LocalProc] ∗pcache;
```

Here, the T[‘X] syntax is used to denote a polymorphic data type. As before, this is just syntactic sugar for *void* pointers: the type variables may only be used indirectly, as base types for pointers.

Figure 3.33: Local process cache

```
struct LocalProc {
    Proc ∗p;
    State ∗shared;
    State ∗local;
    Mono ∗blocks[];
};

LocalProc ∗getLocalProc(Proc ∗);
(State ∗, int) getState(LocalProc ∗);
bool setState(LocalProc ∗);
bool setMacroState(LocalProc ∗, int);
int getBuffer(LocalProc ∗, int)[∗];
```

Figure 3.34: Semi-local layer interface

dictionary with some eviction strategy built-in. Its interface is shown in Figure 3.33. Since this is a strictly sequential data structure, many tricks and optimizations are possible, which are beyond the scope of this chapter. We will simply assume that we have an efficient way to associate local information with process pointers.[8]

The semi-local layer provides **process proxy**, *LocalProc*, objects, which hold local (sequential) information about a process, for use by a single worker thread. Before attempting any modification to a process, the worker first obtains a *LocalProc* pointer to work with, by calling the *getLocalProc* method of Figure 3.35. Proxies offer a transactional view of process states, through an assortment of three functions: *getState*, *setState* (and its cousin *setMacroState*) and *getBuffer*, whose signatures are given in Figure 3.34.

Abstractly, a process proxy is a simple object: a pair made of an **expected** and

[8] For example, one straightforward implementation would be to simply have an array of local payloads in each *Proc* structure. Otherwise, trees or hash tables can be used in conjunction with some kind of doubly-linked list to simulate the eviction protocol.

```
LocalProc *getLocalProc(Proc *p)
{
    var lp := getCached(pcache, p);
    if (lp = null) {
        lp := malloc(sizeof *lp + countof p.args * sizeof *lp.blocks);
        lp.p := p;
        lp.shared := null;
        lp.local := null;
        for (var i := 0; i < countof p.args; ++i)
            lp.blocks[i] := null;
        putCached(pcache, p, lp, freeLocalProc);
    }
    return lp;
}
```

```
void freeLocalProc(Proc *p, LocalProc *lp)
{
    endReadSection(lp.shared);
    freeState(lp.local);
    for (var i := 0; i < countof p.args; ++i)
        endReadSection(lp.blocks[i]);
}
```

```
void allocState(int nvar, int narg)
{
    State *s := malloc(sizeof *s + narg * sizeof *s.descs);
    s.vars := calloc(nvar, sizeof *s.vars);
    s.inds := calloc(narg, sizeof *s.inds);
}
```

```
void freeState(State *s)
{
    free(s.vars);
    free(s.inds);
    free(s);
}
```

Figure 3.35: Concurrent implementation: *getLocalProc*

a **tentative** values. Those correspond to the *shared* and *local* fields of the *LocalProc* structure. The *p* field is just the key to the corresponding process, while the *blocks* array is present purely as an optimization, which we discuss afterward, but otherwise does not affect the abstract semantics of our operations.

getState Fetches a writable copy of the most up-to-date working state for the given process, updating the expected view of the shared state, and executing assistance, if needed. Returns null and a channel index if the process is waiting on said channel and no working state is available.

The most up-to-date working copy is the tentative state, with any previous local modifications, if the shared state is at the expected value (and hence provably older), or a new copy of the shared state, if it has changed since the last call to *getState*.[9] The returned state object, if any, can then be modified, and connected channels can be accessed through *getBuffer*.

setState Compares the actual shared state with the expected value we have read previously, and swaps in our tentative modifications, making them permanent, if the test succeeds. Essentially, just compare-and-swap, with additional handling for dependent read sections, blocks, etc.

getBuffer Returns the current block associated with a given argument if the shared state has the expected value, or null otherwise.

3.8.3 Read section caching

Aside from the above semantics, process proxies also offer some opportunities for optimization, by caching and reusing some costly resources, most notably read sections. As we will see, this method is employed in both *getState* and *getBuffer*.

Instead of calling *safeRead* everytime, we keep a read section open on the last version of the target variable we have loaded. When needed, we simply refresh that read section: by calling *endReadSection* and *safeRead* again only if necessary.

The way this works deserves a little explanation. We recall from the previous chapter that *safeRead* is merely a loop that attempts to match a loaded value with a call to *beginReadSection* that must happen before said load instruction. In essence, we are guessing at the read value and gambling on a read section on it, later aborting and retrying if we realize we are wrong. In the case where we already have a read section open on a previous version of the object, we can skip the first iteration of the *safeRead* loop and jump straight to testing whether we are right in our guess.

[9] The new shared value is considered newer, which is always the case if worker threads only commit at obligatory checkpoints (e.g., when changing blocks), but may be slightly wrong in the presence of non-deterministic early swaps since one thread may progress further with its monotonic updates, while another may decide to commit early and switch to another process, for example. However, unless threads repeatedly call *setState* without making any changes to the state, this strategy is conservative and still allows lock-free progress, since we only discard our own modifications if someone else has made some.

Caching read sections obviously increases the risk of losing more memory if a thread crashes while holding onto those sections. Therefore, the cache and main memory heap need to be adjusted accordingly, so that the algorithm remains non-blocking.

3.8.4 Layered linearizability

Before we continue with more code and linearizability results, let us pause for a moment to discuss the correctness criteria for the layered design. While the channel objects presented so far are small and well-delimited, in this section, we build a full-fledged Kahn process network interpreter. What is then the scope of our linearizable objects?

In fact, it is the process graph itself, as a whole. Indeed, most of the methods we are about to introduce affect multiple components at once. Making progress in a process network requires touching processes and channels in different, non-disjoint combinations. Sometimes, a method may act on behalf of the producer, some other, it may be acting as the matching consumer: same channel, different process objects.

Even in this context, however, the principle of separation still applies. We prove each procedure linearizable in its action on the entire process graph. However, for that purpose, we only need to consider interference from other instances that manipulate the same process and channel objects. Since we present the functions one at a time, it might be difficult to project a full view of the current layer. To ease with mental navigation, each proof starts with the interference hypotheses it relies upon. As a general guideline, the data type descriptions of Sections 3.8.1 and 3.8.2 provide (somewhat implicit) invariants regarding the values and progressions of shared variables, and fields are only modified by a single function or indirectly by others that call it. With this in mind, let us start exploring the upper layers of our Kahn process interpreter.

3.8.5 Reading states

We first study the code of *getState* and its auxiliary function *assistProc*, presented in Figures 3.36 and 3.37. As described above, *getState* acts as a conditional copy from the shared to the local state. Let us prove a few simple properties on these functions.

Lemma 11. *Given on input a* LocalProc *object whose* shared *field has been safe-read from the corresponding* Proc *object, the* assistProc *function is linearizable.*

Proof. We recall that *LocalProc* objects are entirely local to the calling thread. We assume only *assistProc* concurrently modifies the *chi* field in the *State* structure.

First, the function loads the *chi* field from the shared data structure, on line 4. This variable is only updated monotonically from positive or zero to -1. In instances where it reads -1, the function immediately returns, thus linearizes on the load instruction.

Suppose *chi* is not -1. The only actions that do not operate on either local or read-only (after initialization) memory are: the *macroPush* and *macroPop* calls (which are both linearizable due to Lemma 9) on lines 8 and 10, and the shared modification of *chi* on line 12. The invocations linearize at the *macro* call on failure, and on line 12 on success.

```
(State *, int) getState(LocalProc *lp)
{
   if (lp.p.state $\neq$ lp.shared) {
      endReadSection(lp.shared);
      lp.shared := safeRead(&lp.p.state);
      if (lp.local = null)
         lp.local := allocState(lp.p.nvar, countof lp.p.args);
      copyState(lp.local, lp.shared, lp.p.nvar, countof lp.p.args);
      if (lp.local.chi $\neq -1$) {
         // Invalidating the cached block is entirely optional. If not now,
         // it will be replaced the next time we call getBuffer.
         endReadSection(lp.blocks[lp.local.chi]);
         lp.blocks[lp.local.chi] := null;
         lp.local.chi := $-1$;
      }
   }
   var ok, chi := assistProc(lp);
   return ok ? lp.local : null, chi;
}
```

```
void copyState(State *d, State *s, int nvar, int narg)
{
   memcpy(d, s, sizeof *s + narg * sizeof *s.descs);
   memcpy(d.vars, s.vars, nvar * sizeof *s.vars);
   memcpy(d.inds, s.inds, narg * sizeof *s.inds);
}
```

Figure 3.36: Concurrent implementation: *getState*

```
1  (bool, int) assistProc(LocalProc *lp)
2  {
3     bool ok := true;
4     var chi := lp.shared.chi;
5     if (chi $\neq -1$) {
6        var d := &lp.shared.descs[chi];
7        if (d.block = null)
8           ok := macroPop(d, lp.p, chi);
9        else
10          ok := macroPush(d, lp.p, chi);
11       if (ok) {
12          lp.shared.chi := $-1$;
13          chi := $-1$;
14       }
15    }
16    return ok, chi;
17 }
```

Figure 3.37: Concurrent implementation: *assistProc*

Indeed, if the macro operation returns false, then it cannot be satisfied, and a sequential run at that point would produce the same effect.

In the case where it returns true, we know that *macroPush* and *macroPop* invocations on the same descriptor after the first successful instance are idempotent. Furthermore, only *assistProc* sets *chi* to -1, thus the only path that leads to a -1 value passes through a true return from either macro operation. Therefore, instances of *assistProc* that reach line 12 after the first do some redundant work, but otherwise behave exactly as if they had been called at that point, seen the -1 flag and immediately returned. □

Lemma 12. *The* getState *method is linearizable.*

Proof. The three concurrent operations that touch the shared mutable parts of the Kahn graph are the first load on line 3, the following safe read on line 5, and the invocation of *assistProc* at the end.

We note that the call to *copyState* on line 8 copies from a mostly constant (after initialization) state object into a local recipient. The only mutable field in *State* is *chi.* It is however reset to -1 by the following lines regardless of its initial value.

We also remark that if the shared state has changed on the first test, on line 3, then since the expected value is ABA-protected, the same test taken later will produce the same result.

Invocations where *assistProc* returns false linearize at the point of loading or reloading the shared state. Due to the above remark, the test on line 3 taken at the later point, when reloading, would result in the same branching decision. Moreover, since loading

```
bool setState(LocalProc ∗lp)
{
    var ok := cas(&lp.p.state, lp.shared, lp.local);
    endReadSection(lp.shared);
    if (ok) {
        destroy(lp.shared, freeState);
        lp.local := null;
    }
    lp.shared := null;
    return ok;
}
```

Figure 3.38: Concurrent implementation: *setState*

happens before the failing call to *assistProc*, earlier assistance on the same descriptor is sure to also fail with false.

Invocations where *assistProc* returns true linearize either on the first load on line 3, if *chi* is -1, or where the descriptor current when loading the shared state (on line 3 or line 5) is completed (which may occur out of the method). Indeed, at that point the shared state is the same as upon loading, since only completed descriptors can be swapped out. Besides, assistance is guaranteed to succeed, since completion has just happened. □

3.8.6 Updating states

We now tackle the counterpart to *getState*, namely, *setState* (Figure 3.38), which may be called after retrieving and modifying the local state returned by *getState*. Every call to *setState* must be preceded by a matching successful return from *getState*: it is illegal for the client to call *setState* first, or multiple times in a row without an intervening *getState*.

The *setState* function operates mostly as a big compare-and-swap statement, with the ability to introduce a state and its associated descriptor into the shared data structure.

Lemma 13. *The* setState *method is linearizable.*

Proof. The only operation on the shared mutable memory of the Kahn graph is the first and only instance of compare-and-swap. The method linearizes at that point, since every other action happens locally, and thus could be taken at any time. □

There is a straightforward caching optimization illustrated in Figure 3.39. Instead of losing all references to state objects, we instead preventively take a read section on what we believe is going to be the next shared value: our tentative state. Then we, if the gamble is correct, we inline the next call of *getState* into the success path of *setState*, eliminating the unneeded safe read. In the other case, the revert the read section and proceed as usual. Although it is quite simple, we have chosen not to clutter the above proof with these details.

```
bool setState(LocalProc ∗lp)
{
    beginReadSection(lp.local);
    var ok := cas(&lp.p.state, lp.shared, lp.local);
    endReadSection(lp.shared);
    if (ok) {
        destroy(lp.shared, freeState);
        lp.shared := lp.local;
        // Same as in getState.
        lp.local := allocState(lp.p.nvar, countof lp.p.args);
        copyState(lp.local, lp.shared, lp.p.nvar, countof lp.p.args);
        if (lp.local.chi $\neq -1$) {
            endReadSection(lp.blocks[lp.local.chi]);
            lp.blocks[lp.local.chi] := null;
            lp.local.chi := $-1$;
        }
    } else {
        endReadSection(lp.local);
        lp.shared := null;
    }
    return ok;
}
```

Figure 3.39: Concurrent implementation: *setState* (optimized)

```
bool setMacroState(LocalProc *lp, int chi)
{
   lp.local.inds[chi] := 0;
   lp.local.chi := chi;
   var arg := lp.p.args[chi];
   if (arg.kind = IN)
      makePop(arg.ch, &local.descs[chi], &shared.descs[chi]);
   else
      makePush(arg.ch, &local.descs[chi], &shared.descs[chi]);
   return setState(lp);
}
```

Figure 3.40: Concurrent implementation: *setMacroState*

On top of *setState*, we provide the *setMacroState* function (Figure 3.40), which prepares the local state for a macro operation (*macroPush* or *macroPop*) and attempts to swap it in. A call to *setMacroState* takes the place of a similar invocation of *setState*, with regard to alternation with *getState.*

Corollary 5. *The* setMacroState *function is linearizable.*

Proof. The shared descriptors are constant and accessed read-only. Everything else is local. Thus, every invocation linearizes at the call to *setState.* □

3.8.7 Accessing data

Lastly, for the semi-local cache, we study how to access data, using *getBuffer* from Figure 3.41. Similarly to *setState*, every call to *getBuffer* must be preceded by a matching successful instance of *getState.*

Lemma 14. *The* getBuffer *method is linearizable.*

Proof. There are two statements that access the shared mutable state of the graph: the test against the shared value of the data array on line 5, and the call to *getMono* on line 7.

As above, if the invocation hits the cache and proceeds without branching in, then it linearizes at the only contended access, on line 5. (Note that it cannot return null, due to the way the macro queue is structured; see Figure 3.26.)

Otherwise, it calls *getMono* on line 7, which returns either null if the descriptor is not current anymore, or a valid block otherwise. In both cases, the invocation linearizes on line 7. Indeed, since we hold a read section over the block locally obtained on line 3, it cannot be swapped back in between line 5 and line 7. Therefore, performing the test sequentially together with the rest of the method at the later point yields the same result. □

```
int getBuffer(LocalProc *lp, int chi)[*]
{
    var *a := lp.blocks[chi];
    var d := &lp.shared.descs[chi];
    if (d.ch.data[d.turn % 2] $\neq$ a) {
        endReadSection(a);
        a := getMono(d, lp.p, chi);
        lp.blocks[chi] := a;
    }
    if (a = null)
        return null;
    else
        return (int[*])(d.ch.len * 3, a.buf);
}
```

Figure 3.41: Concurrent implementation: *getBuffer*

3.9 Non-blocking transitions

After much preparation, the next step in our journey finally brings us back to high-level design considerations first introduced in our sequential overview at the beginning of this chapter. We now look at how transitions are realized using the tools we have just built.

We recall that transitions are made of two halves: the start and the end half-transitions. The former consists in updating the process state with the result of the *step* function. The latter completes the transition by executing a channel input or output order triggered by the first half.

In our implementation, **checkpoints are deterministic** and reduced to the minimum necessary: they occur only when a process needs one of its monotonic buffer replaced, or has reached its final state. The succession of shared states is therefore the same for all worker threads. In particular, if *setState* or *setMacroState* fails, then we are guaranteed that any pending local operation has already been completed on the global state, i.e., that the current thread is late.

3.9.1 Start half-transitions

The start half-transition, Figure 3.42, plays out similarly to its sequential counterpart: it applies the *step* function to the current state and handles termination and reconfiguration. The main differences are:

- the working state being a copy, for *step* to write into;
- reconfiguration requiring compare-and-swap, so we do not replace a subgraph multiple times;

```
 1  int startHalf(LocalProc ∗lp, State ∗s)
 2  {
 3      var vs := (int[∗])(lp.p.nvar, s.vars);
 4      if (not lp.p.step(lp.p.args, vs))
 5          return 0;
 6      if (lp.p.reconfigure ≠ null) {
 7          var conf := lp.p.reconfigure(lp.p.args, vs);
 8          if (not cas(&p.conf, null, conf))
 9              freex(conf);
10      }
11      s.final := true;
12      return setState(lp) ? 0 : −1;
13  }
```

Figure 3.42: Concurrent implementation: *startHalf*

- no apparent counting of finished children.

Another notable feature is the return value, which we will encounter again when dealing with end halves. Instead of the customary true–false dichotomy, here we have three status values: zero indicates normal completion, as is usual in C, a positive value signals that we have been too fast and need to wait (listed here for completeness but only used for *endHalf*), while a negative one means we have been too slow and another worker has already completed the current half-transition. This is useful information for our future scheduler.

Lemma 15. *Assuming it is called on a local state with no pending end half-transition,*[10] *the* startHalf *function is linearizable.*

Proof. If *step* returns false and the function branches out after line 4, the process has not finished, and we return zero. The computation is entirely local, so we can linearize anywhere, for example, at the call.

Else, *step* returns true and the routine branches into either reconfiguration if *reconfigure* is not null, or single-process termination otherwise. In both cases, a new final state is swapped in by calling *setState* on line 12, which is linearizable by Lemma 13. We assume that *startHalf* is the only method to introduce final states or new configurations.

The compare-and-swap statement on line 8 may either succeed or fail, with the same overall effect. Since we are in the only procedure that modifies the *conf* field, reaching line 12 always guarantees that *conf* is non-null and—by determinism of the Kahn process network and the *reconfigure* function—contains an equivalent value representing the subgraph.

[10] Otherwise, the behavior of the *step* user transition function is ill-defined.

If *setState* succeeds, then the shared state has not changed at that point and the whole *startHalf* invocation can linearize at it. If it fails, then the shared state has changed, and we can also linearize at the *setState* invocation, since a *startHalf* run beginning at that point would simply end up attempting the same *setState* call and similarly failing. □

3.9.2 End half-transitions

End half-transitions complete the transition by executing any channel command requested by the start half. It is the only routine that uses and changes the *inds* array (locally, of course) to track consumption and production into the monotonic buffers, so they can be used as queues. We apply the ideas developed previously: semi-local operations on monotonic blocks (*monoPush* on line 16, *monoPop* on line 14), separated by obligatory checkpoints on saturation, in the form of shared state changes (*setMacroState* on line 10).

It should be noted that *endHalf* actually does not always complete the end half-transition exactly, due to the way macro operations play out. If the requested channel action requires a block change, then the procedure returns after setting the new state, without actually doing the work. Although the exit status is zero, the half-transition is still pending; it will only close after another call to *getState* and *endHalf*.[11] This does not affect linearizability, however; the same quirk would manifest should we decide to run the function sequentially.

Lemma 16. *In all histories made of invocations of* startHalf *and* endHalf *(with the implicit* getState*), the values written to the monotonic buffers are indeed monotonic, i.e., different threads writing to the same location within a buffer write the same value.*[12]

Proof. First, we remark that only *endHalf* modifies locations within buffers, through *monoPush* and *monoPop*. Furthermore, *endHalf* invocations under the same expected shared state (as loaded by the call that provides s to *endHalf*) access the same buffers.

We prove the lemma by induction on the length of histories. By linearizability, we consider whole invocations of *startHalf*, *setMacroState*, *monoPush* and *monoPop* as atomic actions.

If the history is empty, no buffer is written, therefore the property holds. Suppose it is true for all histories of length less than or equal to n. Let H be a history of length $n + 1$.

[11] We thought long and hard about a simple way to have *endHalf* actually guarantee a full half-transition; however, the added complexity was judged counterproductive, hence this slight but unfortunate deviation.

[12] This lemma looks very much like Theorem 1, but is in fact weaker; it only shows that within a same concurrent execution, the same values are produced and the same decisions taken by any redundant segment of a given process. It says nothing of different scheduling choices; in a sense, it states the opposite: that there are no choices to be had, and all workers are bound to repeat the same program for every process (which is good). We use it to prove linearizability of the *endHalf* procedure, which will be used to build an equivalence with sequential schedules later on, but for now, this is a simpler lemma, with a simpler proof.

```
int endHalf(LocalProc ∗lp, State ∗s)
{
    var chi ≔ s.vars[0];
    var vi ≔ s.vars[1];
    var buf ≔ getBuffer(lp, chi);
    if (buf = null)
        return −1;
    if (s.inds[chi] = lp.p.args[chi].ch.len) {
        s.inds[chi] ≔ 0;
        return setMacroState(lp, chi) ? 0 : −1;
    }
    bool ok;
    if (lp.p.args[chi].kind = IN)
        ok ≔ monoPop(buf, s.inds[chi], &s.vars[vi]);
    else
        ok ≔ monoPush(buf, s.inds[chi], s.vars[vi]);
    if (ok) {
        s.vars[0] ≔ −1;
        s.vars[1] ≔ −1;
        ++s.inds[chi];
    }
    return ok ? 0 : 1;
}
```

Figure 3.43: Concurrent implementation: *endHalf*

If the last action is not a successful instance of *monoPush* or *monoPop*, then nothing new is written to any buffer, therefore the property holds by induction hypothesis on the n-prefix.

Otherwise, the last action ξ is a successful invocation of *monoPush* (line 16) or *monoPop* (line 14) within an invocation of *endHalf*. If this is the first such call at this index in the buffer, then the property trivially applies. Else, there exists another action ξ' of the same nature that has written a value at this index before in H.

- Suppose ξ and ξ' are based off the same expected shared state s'. Local states can only be constructed by successive applications of *startHalf* (through the *step* function) and *monoPop*. The initial shared state used by both derivations is the same, hence the buffers are also the same by the remark above. At each successive step, by induction hypothesis, any values read by *monoPop* must also be the same. Therefore, it takes the same steps to reach the local state of both ξ and ξ', which must be equal. Thus, they write the same value.

- Suppose ξ and ξ' are based off different expected shared states. ξ must have an older expected state s, as newer states would have a higher index, or a different buffer, depending on whether *setMacroState* on line 10 is called with the same *chi* or not. Let us consider any older derivation from the introduction of s to its replacement by the next state. This sequence must exist since ξ' is newer and is based off a subsequent expected state. Either the derivation is longer than the number of steps required to reach ξ, or it is shorter.

  If it is longer, then by the same reasoning as above, starting from the same initial state and with the same input and transitions, it hits a modification of the same location as ξ and ξ' after the same amount of steps used for ξ. By induction hypothesis on a prefix of H, the value written is the same as that of ξ' and ξ.

  If it is shorter, then we should have hit the same *setMacroState* call before reaching ξ, which is impossible.

In every case, ξ writes the same value as ξ'. The property applies at rank $n + 1$. □

Corollary 6. *Checkpoints are deterministic.*

Proof. As a straightforward extension of the above proof, since local states based off the same expected shared state are equal after a given number of steps, then they all attempt to establish a checkpoint after the same number of iterations, when a buffer needs to be swapped. □

Lemma 17. *Assuming it is called on a local state with a pending end half-transition, the* endHalf *function is linearizable.*

Proof. If an invocation fails after loading the buffer on line 5, then it means the state has changed between the call and *getBuffer* (linearizable, due to Lemma 14). We can linearize at *getBuffer*.

Otherwise, either the block is saturated on the side of the current process, and we call *setMacroState*, on line 10 (linearizable by Corollary 5), or it is not, and we perform a monotonic operation on lines 14 and 16 (linearizable by Lemma 5).

If *setMacroState* succeeds, then the shared state has not changed at that point, and so previous tests would pass identically were the invocation to take place at that instant instead. If it fails, then it does not affect the shared structure occurs; the shared state, however, has changed from its expected value. In the corresponding linear run, the previous *getBuffer* test would branch out, yet also return -1. Therefore, in both cases, we linearize on line 10.

Finally, if we go down the route of monotonic operations, then we call either line 14 or line 16, depending on its nature (which is constant after initialization, being drawn from the *args* array). Since checkpoints are deterministic by Corollary 6, we know that if the call fails, no new shared state has been published at that point; otherwise, the current operation would have been completed first. We can thus linearize at the monotonic operation.

If the monotonic operation succeeds, then there are two cases: either the expected state is unchanged at the call, in which case we simply linearize at that point, or it has changed by the time we invoke *monoPush* or *monoPop*. By Corollary 6, the current operation must have been completed by another thread before reaching the next checkpoint. Therefore, we can pick as our linearization point the instant just before the next shared state is installed by the other thread (which lies outside of the current method invocation). At that point, the operation is sure to succeed (albeit redundantly so), yet the state remains unchanged, thus the test on line 5 passes and yields the same buffer as in the current execution. □

3.10 A non-blocking interpreter

At long last, we can finally write a new concurrent lock-free implementation of the *cocall* function introduced early in this chapter, in Figure 3.44. It makes use of alternating *getState*, *startHalf* and *endHalf*, as we have outlined already in Lemma 16.

Lemma 18. *The* cocall *function is linearizable.*

Proof. Note that *getLocalProc* is purely a sequential helper routine that does not touch shared memory.

Invocations that return early after loading the state on line 5 linearize at *getState* (linearizable by Lemma 12), since the tests are totally local.

If we branch out after the call to *startHalf* on line 11, there are two cases: either *startHalf* returns zero or it returns a negative value. If zero, we can linearize at *startHalf*, since the state has not changed at that point, hence a call to *getState* in the linear run would keep us on the same path. If negative, then we can also linearize at *startHalf*, since a sequential run at that point would simply return true–null earlier, when testing for the *final* field on line 8.

```
(bool, Chan *) cocall(Proc *p)
{
    var lp := getLocalProc(p);
again:
    var s, chi := getState(lp);
    if (s = null)
        return false, lp.p.args[chi].ch;
    if (s.final)
        return true, null;
    if (s.vars[0] = $-1$ or s.vars[1] = $-1$) {
        if (startHalf(lp, s) $<$ 0 or s.final)
            return true, null;
    }
    assert(s.vars[0] $\neq -1$ and s.vars[1] $\neq -1$);
    chi := s.vars[0];
    var status := endHalf(lp, s);
    if (status $<$ 0)
        goto again;
    return status = 0, lp.p.args[chi].ch;
}
```

Figure 3.44: Concurrent implementation: *cocall*

Otherwise, the invocation reaches line 16. If we do not return from *startHalf* or before, then the process has not finished and is not waiting for a macro block either, just before the call to *endHalf*. Very importantly, it implies that *startHalf* did not touch shared memory. There are two cases:

- If *endHalf* returns zero or a positive status, then the state has not changed at *endHalf*, so a displaced *getState* at that point would yield the same result. It is the linearization point.
- If *endHalf* returns a negative value, then the channel buffer has changed, hence the shared state with it. This particular call to *endHalf*, hence this entire *cocall* iteration, has had no effect. Indeed, since the expected state has moved, the next call to *getState* will discard any tentative modifications we have just made. Therefore, we can simply loop on the *cocall* procedure and linearize at the next opportunity. □

Due to the caveat discussed above concerning the *endHalf* procedure, this concurrent version of *cocall* does not have the exact same semantics as the sequential *cocall*. However, it does accurately report waits (false–channel return pair), which becomes important for its use in *copoll* later on.

Nevertheless, in a concurrent execution made of *cocall* invocations, we can construct a **compatible** legal sequential schedule for the same Kahn process network, based on this altered semantics.[13] We consider the linear history of *cocall* equivalent to the concurrent one. Then, by Lemma 16 and Corollary 6, we know that every half-transition is either the first to hit such an channel index or position in the process program, or it is redundant with a previous such half-transition that has already been carried out by a prior invocation of *startHalf* or *endHalf*, hence of *cocall*. We build the compatible sequential schedule as follows: if the concurrent half-transition is first (and even if it is purely local), then we append the corresponding sequential half-transition to the compatible schedule; if not we do nothing.

Lemma 19. *The compatible schedule is legal.*

Proof. Sequentially, we verify that a succession of successful invocations of *startHalf* and *endHalf* on non-repeating local states implements a Kahn process network using queues of queues and a process tree not much different from the sequential interpreter.[14] Our previous construction builds such a non-repeating sequence by picking only the first half-transition at each index. By Lemma 16 and Corollary 6, we know that later redundant half-transitions are entirely idempotent and do not spoil the construction from those that came first. □

[13] The only problem is that we need to look at the actual effects on shared memory, at the level of states and buffers, rather than basing our scheme entirely on the observable return values from *cocall* or *endHalf*; in a sense, we are implicitly using an additional (unnamed) interface that gives us a glance at the shared data structure. We must do this to distinguish between unfinished end half-transitions and those that actually completed, following an invocation of *endHalf*.

[14] Although we give no actual proof, it is really just a complicated sequential implementation with compare-and-swap statements that cannot fail, and queues of queues instead of regular ring buffers.

Corollary 7. *For all local states observed between calls to* cocall*, there exists a prefix of the compatible schedule where an equivalent (same process variables, same production and consumption indices) state was current.*

Proof. From the construction of the compatible schedule. If the state comes from an effective half-transition (that was first) that introduces a corresponding sequential half-transition, then the prefix is the schedule up to and including the introduced sequential half-transition. This prefix builds the same state when considering only non-redundant half-transitions, as seen in the proof of Lemma 19.

Else, the state comes from a redundant half-transition, by Corollary 6, and there exists an equivalent first half-transition, which has a compatible prefix. □

3.10.1 Termination

Since start transitions do not take care of counting finished processes anymore, we need another way to check for termination. This is accomplished by the *checkFinished* procedure, shown in Figure 3.45. It should be called on processes whose children are suspected to have finished themselves. This task is best left to a smart scheduler (with some help from the core functions presented in this section). For now, however, let us simply observe that by default, it suffices to chain *checkFinished* after any true–null return from *cocall*, to guarantee that termination will be detected eventually (albeit inefficiently).

Lemma 20. *The* checkFinished *function is linearizable.*

Proof. As we have seen many times now, branching out at either of the first two tests following line 3 means a linearization point at the call to *getState*. Indeed, the tests only touch constant (at that point) or local memory.

Going further, if *update* is false, then the invocation linearizes on line 8, when loading *njoined*. At that point, we are in a final state, which means the first test cannot fail, and *reconfigure* is constant after initialization, so does not change in between either.

If *update* is true, then we can pick the last iteration of compare-and-swap on line 16 as the linearization point. The same reasoning as in the previous paragraph applies; in addition, the last compare-and-swap also provides the only (optional) modification, and the value of *nj* used in the final comparison when deciding the return value. Therefore, everything works as if we had executed instantly at that point. □

3.10.2 Assemblage

We can now plug everything into the framework we have built earlier, more specifically the *copoll* method, Figure 3.46.

The code of concurrent *copoll* is similar to its sequential counterpart, except the invocation of *cocall* now references the concurrent version of *cocall*, which, as detailed above, has slightly different semantics. The main result of this chapter, which should now become apparent, is that, in spite of this underlying difference, *copoll* itself is fully linearizable with respect to the specification induced by sequential *copoll*. Said specification is certainly not trivial.

```
bool checkFinished(LocalProc ∗lp, bool update)
{
    var s, _ := getState(lp);
    if (s = null or not s.final)
        return false;
    if (lp.p.reconfigure = null)
        return true;
    var nj := lp.p.njoined;
    if (update) {
        int i;
        for (i := nj; i < countof lp.p.conf.procs; ++i) {
            if (not checkFinished(lp.p.conf.procs[i], false))
                break;
        }
        while (nj < i) {
            cas(&lp.p.njoined, nj, i);
            nj := lp.p.njoined;
        }
    }
    return nj = countof lp.p.conf.procs;
}
```

Figure 3.45: Concurrent implementation: *checkFinished*

```
int copoll(Proc ∗p, bool w[])
{
    for (;;) {
        var q := schedule(p, w);
        if (q = null)
            return −2;
        var ok, ch := cocall(q);
        if (ch = null) {
            if (checkFinished(q)) {
                if (q = p)
                    break;
            }
        } else if (not ok) {
            for (var i := 0; i < countof w; ++i) {
                if (w[i] and p.args[i] = ch)
                    return i;
            }
        }
    }
    return −1;
}
```

Figure 3.46: Concurrent implementation: *copoll*

First, its interface involves multiple helper routines (e.g., *mkproc*) around the main *copoll* method. Those serve to abstract input and output values. Most importantly, we recall that the *Proc* object is not expected to be manipulated by the user directly. Instead, it is to be considered an opaque structure defined by the arguments passed to *mkproc.* This introduces a simple equivalence between the input arguments of sequential and concurrent *copoll.* There are other ways around this. We could, for example, fully separate *Proc* and *Chan* into user-facing immutable structures and internal mutable objects; however, this comes at the price of added indirection and implementation complexity.

Second, we recall that the expected behavior of the interpreter is defined in terms of allowed interleavings of call and return actions from its methods. In the case of *copoll*, concurrency has an influence on the end result: this is modeled in the sequential specification by the *schedule* black-box procedure, which non-deterministically returns a process to execute. In doing so, we mask away the potential environmental interactions between the interpreter object and the system, thus allowing for the linearization of *copoll.*

In the implementation of concurrent *copoll*, notice how unlike the sequential *copoll* method, we have no way of testing for empty or full channels without examining the attached processes. Therefore, this version does not start with a scan of the *w* array (compare Figure 3.11). Instead, we rely exclusively on the scheduler eventually giving us a process waiting on half-bound channel to be supported by the environment.

This approach is, evidently, highly inefficient. As with termination, this should be solved by better scheduling techniques that keep track of information and can answer such queries as: "Are such and such channel preventing processes from progressing?"

Theorem 2. *Plugging our concurrent implementation of* cocall *into* copoll *yields a concurrent, lock-free, disjoint-access-parralel interpreter.*

Proof. As we have seen, even though concurrent *cocall* is not equivalent to sequential *cocall*, at the very least, it accurately reports waits. Therefore, when plugged into *copoll*, which simply iterates on any non-wait return code, such internal fluctuations are ignored, and the external behavior complies with the specification, which can be described as follows.

Every call to *copoll* appears as if it linearizes at the point where it finds a suitable channel to return to its caller. Any progress it makes on Kahn processes to get there are seen as the effect of some schedule obtained non-deterministically—it is in fact the result of the scheduler plus the many hazards of concurrency.

This implementation is lock-free, since any loop ensures global progress. The one in *cocall* is entered only if a buffer has been replaced, and the main loop in *copoll* calls *cocall* with each iteration. Through the various lemmas of this chapter, we have seen that *cocall* builds upon pieces that guarantee an accurate representation of the Kahn process network, both at the local and shared level (although they may indicate different times, they all belong to the same equivalent sequential schedule as shown in Corollary 7).

Finally, it is disjoint-access-parallel through the use of blocks and descriptors, such that thread executing a specific process only competes for resources associated with its immediate actions: the shared process state, and connected buffers. □

3.10.3 Lock-free performance

Although we must admit that, at the time of writing, this non-blocking interpreter algorithm has yet to be fully implemented and tested in a real-world environment, we want to summarize and justify some of the motivations behind it, and why we believe it has the potential to be a robust, fast and realistic implementation.

The main appeal of our work lies in the fact that it combines lock freedom with characteristics that are typically good for performance.

- First, we would like to reiterate that it mirrors the disjoint-access parallelism of the source Kahn process network: a transition, as executed by *startHalf* and *endHalf*, only touches the parts of the data structure corresponding to the specific process and channel that it covers.

- Second is the relatively low overhead of transitions in terms of atomic operations. Read-modify-write operations are only involved when updating the shared process states and channel buffers (one compare-and-swap instance for each), or when terminating. Most normal channel input and output use only simple read and write operations.

  Very importantly, contrary to some popular inter-thread communication devices that do transfers in large chunks at a time, we push and pop items one at a time, and therefore do not constrain the original process network in any way, with regard to the length of its cycles.[15]

  This, however, is distinct from the fact that it is possible to increase the size of channels, even beyond what can naturally be used by the Kahn process. In our implementation, this leads to less frequent shared state updates, thus less expensive compare-and-swap instructions. We thus offer a convenient way of trading memory for less overhead, which often has a sizable impact on practical performance.

- Although we rely heavily on the presence of a lock-free memory management system,[16] which is often criticized for the high cost associated with taking read sections, our algorithm also offers a way mitigate this expense: through the caching mechanism of the semi-local layer. When cached, it only costs one simple load instruction check whether the pointer is still valid; we could even ditch this test completely, at the risk of doing more redundant work. Again, it is possible for the user to determine how much memory to trade for potentially unused cached read sections.

While caching and local execution are great tools to have, we would be right to question whether it is wise to promote them as a way to combat overhead. After all, using very long channels would cause processes to be run for correspondingly long times before

[15] We recall that in a process network with feedback loops, it may sometimes be necessary to produce before being able to consume more (and vice versa), therefore it is not always possible to wait to have more data to write to an output channel, or read from an input channel.

[16] As do all complex lock-free data structures.

threads have to compete for a checkpoint, therefore leading to potentially tremendous amounts of redundant work.

However, we feel that this is best addressed as a scheduling problem; after all, nothing prevents our interpreter threads from cooperating in other, less costly (and less authoritative), ways. For example, we could easily imagine a system of priority whereby each worker flags the process or processes it wants to execute, and others refrain from doing the same, unless they have no choice.

This brings us to the topic of scheduling, which we left as a black box in our final *copoll* implementation in Figure 3.46.

3.10.4 Scheduler independence

In our concurrent interpreter, following the same design philosophy that we used in the sequential case, there is an almost complete separation between the core interpretation mechanisms (under *cocall*), and the process scheduler (represented by the *schedule* routine). In particular, progress will be made with any sufficiently fair scheduler, as we have seen in Section 3.2.3.2.

Most notably, this allows us to use scheduling techniques that may be slightly incorrect in a lock-free environment, by interleaving their decisions with a fair scheduler based on traversal of the process graph, as discussed in that section. Concretely, we support two kinds of deviations from correct fair scheduling behavior:

- process duplication, where two worker threads schedule the same process for execution at the same time;
- and process loss, whereby every thread forgets about some process that is never selected again.

Moreover, the fair interleaving trick of Section 3.2.3.2 ensures that no process is ever completely forgotten. We thus head into the last chapter of our construction with considerable flexibility as regards the remaining major component of the interpreter, namely, scheduling.

Chapter 4

Dynamic scheduling in shared-memory systems

In this chapter, we turn our attention to practical matters: realistic implementations and applications of Kahn process networks. In particular, we discuss scheduling, in its various forms.

As a starting point, in Section 4.1, we present solutions to the problem of process scheduling, mentioned several times in the last chapter, only to be invariably relegated to a background role. The non-blocking interpreter routines we have built—hereafter referred collectively as the **core interpreter**—only make up half of the equation, half of the ingredients needed for good lock-free performance. The other necessity is a good process scheduler.

In Section 4.2, we take a lower-level view. We examine how our algorithms fare when mapped onto real hardware threads. Indeed, we have assumed until now a sequentially consistent world, made of pure interleaving, which by modern standards can be considered unrealistic. Precise hardware and language memory models exist but involve more complex rules and require special handling, such as adding memory fences or specific memory ordering tags to instructions. We thus look at what is needed to keep ourselves safe and correct in those environments, and how the added machinery affects the performance characteristics of our algorithms.

Finally, we wrap up our study by adopting a slightly different angle: that of applications. Section 4.3 deals with several programming patterns that can be used with Kahn process networks, and more generally how computations can be designed for efficient execution as process networks, and on our proposed implementation in particular. We discuss how they benefit from or are hindered by various strategies devised in the rest of the chapter, or sometimes more pervasive mechanics inherent to our approach.

4.1 Process scheduling

The interpreter we have built, in the previous chapter, is very reliant on the existence of a function that performs **process scheduling**. In our design, we split the life of a

Kahn process into transitions between states. In short, a transition takes execution (of a single process) from one channel input or output operation to the next. Core interpreter routines alternate with the scheduler after each transition, giving us the opportunity to reevaluate our priorities and decide on a follow-up: whether it be pushing the same process further, switching to another one, or maybe even waiting (for a bounded amount of time, to stay lock-free). Process scheduling is the brains that drive all of these things, it is the essential procedure that answers the simple question: “What do we do next?”

To be more precise, as we have seen, transitions can be complete or incomplete, where only one half or none at all gets done before returning to the scheduler. In this chapter, we generally spare ourselves the trouble of distinguishing between all the subcases when looking at the bigger picture, unless they contribute something essential to the discussion—which usually they do not.

In the simplest case, the whole of scheduling can be summed up in a single *schedule* function that returns the next task to accomplish. However, as we will see, more involved strategies often require the use of hooks, i.e., code additions to existing procedures whose sole purpose is to collect information that will guide the decision process. As a rule, we will always make sure that these insertions do not otherwise alter the semantics we have previously established outside of what is covered by the non-deterministic nature of *copoll.*[1]

4.1.1 What makes a good scheduler?

There are several qualities we can look for in a process scheduler. While it is possible (and in some sense desirable, see Section 3.2.3.2) to pick targets at random, in general, it is inefficient. Lock freedom implies an ability on the part of worker threads to execute any process at any time, but it does not mean we should waste time scheduling a same process in different threads at the same time, if given the choice. As a rule of thumb, fault tolerance is a constraint, and as such incurs a performance penalty. Therefore, in the general case, we are much better off trying to emulate an unsafe—blocking—system as much as possible.

The question then becomes: what measure of unsafety is acceptable, for a process scheduler? As we recall from our prior introduction to the subject, Section 2.1.1, lock freedom should not be considered in a vacuum. We want non-blocking objects that linearize to useful specifications. In this subsection, we thus discuss what such a “good specification” entails.

4.1.1.1 Partitioning

In the last chapter, we saw how to characterize a sequential scheduler, in terms of maximal progress and fairness, applied to a function over the whole network state. Unfortunately, these do not translate very well to a concurrent context. Indeed, let us imagine that *schedule* is a pure function of the current state of the Kahn process network, i.e., of

[1] From the point of view of software engineering, they are transverse, or, in aspect-oriented terms, cross-cutting, similar to profile or debug instrumentation.

the individual process states and channel contents. Then calling *schedule* multiple times with the same input should produce the same output—the choice of the next process to execute. In that case, a linearizable concurrent version would instruct multiple worker threads toward the same task if invoked at close intervals, before any thread has the opportunity to update the network. This, in general, is undesirable since we want to take advantage of the disjoint-access properties we have worked so hard to achieve. For those to yield any benefits, however, workers need to operate on different processes simultaneously. Otherwise, we would simply end up doing redundant work.

Looking at the problem from a different angle, suppose we forget about non-blocking constraints, for a second. What would be the best schedule if all threads were guaranteed to execute at the same speed without delays, for example? In these conditions, we should strive to give each thread separate but equivalent amounts of work. It is inherently a classic **partition optimization** problem: given a set of tasks to accomplish (acceptable process transitions) with associated weights (the time needed for the computation), split the work into subsets such that the subset sums match as closely as possible. Even under our ideal no-delay same-speed hypotheses, the problem is further complicated by a couple of factors, however:

- As soon as one task completes, the set to partition may change. Recall that one transition equals one channel operation. Thus, we may observe up to two two changes: the executed process may or may not be able to undergo an immediate successive transition, and its peer on the other side of the channel may be able to run now if it was waiting before.
- Accurately putting a time estimate on each task is a very hard problem on its own, requires a cost model, as well as deciding what kind of estimate it is that we want. Do we balance for the average? Do we shoot for the worst-case execution time? This becomes downright impossible to achieve in any automatic capacity, in practice, in the case of a run-time library such as ours, where the transition functions are given as black-box user-defined procedures.

To summarize, without further information, provided by the user, or some higher-level compiler, perhaps, there is little we can do, as a mere run-time library to figure out the contents of a potential transition, short of attempting to execute it.

It should be noted that there exist restricted classes of Kahn process networks, such as the synchronous data-flow graphs of Lee and Messerschmitt [1987] and its successors, that exhibit more regular properties which can aid in scheduling. Moreover, given more control over the source Kahn process programs (e.g., in a form of a compiler), there is no doubt that automated processing could be conducted to yield suitable hints (e.g., duration estimates or channel affinities), for example by applying well-known control-flow and data-flow analysis techniques borrowed from the field of compiler construction. There are also several data-flow languages that could be considered as candidates for a higher-level representation, one that would allow scheduling information to be collected and handed down to our run-time library in some form or another. Among those, perhaps the most promising option points toward equational synchronous data-flow languages in the tradition of Lustre [Caspi et al., 1987],[2] which translate nicely to Kahn process networks, as it is, and could potentially be integrated to provide more cues with regard to parallel scheduling in the future. Alternatives include declarative–imperative hybrids a la StreamIt [Thies et al., 2002], which already offer good compile-time scheduling abilities for restricted subsets of process networks. Looking further, we could imagine hypothetical adaptations and extensions of the more distant family of task-based parallel languages (of which there is no shortage, e.g., the OpenMP standard in its fourth version, StarSs [Planas et al., 2009], OpenStream [Pop and Cohen, 2013], or Swan [Vandierendonck et al., 2013], to name a few) or distributed frameworks that feature channel constructs in the vein of the widely known MPI library.

Those all constitute interesting research directions for the future. However, in the present work, we concentrate on the minimal case we have described: our Kahn process implementation works as a run-time library embedded into a host language, where transitions are represented as functions written in that same host language. While we may thus forgo some cleverness as regards scheduling, this decision is not without its own merits. The simplicity of its interface makes it possible for programmers to write Kahn processes as state machines directly in the language of their choice, e.g., directly in C, with all the convenience (and inconvenience) it brings. On the other hand, requiring the complete scheduling information required to compute a good partition of the tasks at hand is likely to prove prohibitively complex for human users.

In practice, from our experience, even just writing the state machines in raw C already amounts to a rather specialized task. We can only imagine how much harder it could become were we to add compulsory control-flow and data-flow analyses to the mix without the assistance of a compiler. More than likely, the cost-efficiency would plummet to unacceptable levels well before the point where any sizable Kahn programs could be written.

4.1.1.2 Still more waiting

Given our (self-imposed) constraints, all potential actions are a priori identical. It would seem that the best we can do is simply to randomly partition the work, assuming uniform weights, and give it to threads. Does it mean we are back to square one?

While, this would be true if we had to decide upfront where to allocate time, we have already established that this is not the case. Quite to the contrary, our system is built around a form of online scheduling, also known as **dynamic scheduling**: after each transition, control comes back to the scheduler. Therefore, it is possible to correct any imbalances after the fact, or at least, we can attempt to. The basic blocking algorithm thus implements a straightforward greedy strategy, as illustrated by Figures 4.1 and 4.2.

This process scheduler is somewhat basic and abstract. It adapts to tasks of different lengths, thanks to letting them sit in the A set and not reassigning them immediately. However, beyond that, it does not, for example, distinguish between complete and in-

[2] Not to be confused with the above usage of *synchronous data-flow* by Lee and Messerschmitt [1987].

We start with a set W of tasks to assign to worker threads, and another set A of currently assigned tasks and their hosts. F contains processes that have terminated. When asked for something to do by a thread T, through *schedule*:

```
getTask():
    **While** $W = \emptyset \wedge A \neq \emptyset$
        Wait
    **If** $W = \emptyset$
        // The process network has terminated.
        End
    **Else**
        $x' \leftarrow$ some random task in $W$
        $W \leftarrow W \setminus \{x'\}$
        $A \leftarrow A \cup \{(T, x')\}$
        **Return** $x'$.
```

Figure 4.1: Basic blocking scheduler: *getTask*

When a worker is done with a process, it tells the scheduler to make it available for others again:

```
putTask($x$):
    $A \leftarrow A \setminus \{x\}$
    **If** $x$ is finished
        $F \leftarrow F \cup \{x\}$
    **Else**
        $W \leftarrow W \cup \{x\}$
```

Figure 4.2: Basic blocking scheduler: *putTask*

```
1   putTask($x$):
2       $A \leftarrow A \setminus \{(T, x)\}$
3       If $\exists u,\ [x$ has just completed a transition on $u]$
4           If $\exists y,\ (y, u) \in D$
5               $D \leftarrow D \setminus \{(y, u)\}$
6               $W \leftarrow W \cup \{y\}$
7           $W \leftarrow W \cup \{x\}$
8       Else if $\exists u,\ [x$ is waiting on $u]$
9           $D \leftarrow D \cup \{(x, u)\}$
10      Else
11          $F \leftarrow F \cup \{x\}$
```

Figure 4.3: Blocking scheduler with channel waits

complete transitions. What happens if a thread returns a process to the pool simply because its channel operation cannot be carried out just yet? It is simply returned to the W pool, and thus may end up being chosen again before it is ready to make any progress. This naive strategy is sometimes termed **polling** or **active waiting**, due to its resemblance to a busy loop: we simply poke things in the pool hoping to fish out the good one, the one that will bring the whole network one step forward. Needless to say, if dormant processes—those waiting on channels—outnumber the suitable ones, this can be a recipe for disaster. It might not be incorrect, but it could be horribly slow.

To help with these situations, we introduce the new D work set, as shown in Figure 4.3, which contains pairs mapping a waiting process to the channel it is waiting for. Since Kahn processes can only block on a single channel at a time, D should have at most one pair for each process.

Notice how the W, A, D and F sets are mutually exclusive. Processes flow from one to another depending on their current status. At any point in time, only tasks in W are eligible for execution by a worker. Conversely, every process in W has at least a half-transition to offer. Indeed, initially, all processes can at least complete a start half-transition. Afterward, whenever a transition goes incomplete, the corresponding process is put into a D set (line 9), and only exits that state when the opposite actor makes a move on the connecting channel (line 5); at that point, the process goes back to W with at least one end half-transition ready.

4.1.1.3 Moving between work sets

From the code in Figure 4.1, we can clearly see where the source of blocking behavior comes from: if all the work is currently assigned to other threads, then we wait. Could we do it without waiting? The intuitive solution is to get tasks from A if W is empty. However, if the process is paired in A, it means some other worker is already taking care of it. In an ideal fault-intolerant world, this has a couple of implications.

If we know that some other thread is working on it and has started before us, is there really a need to jump onto the same transition? If threads are guaranteed to progress at the same speed, the answer is obviously negative: since Kahn transitions are deterministic, we would spend as much time as any other execution unit, thus we cannot catch up if they started early. If some mild delays are allowed, the question becomes more nuanced; some external factors (e.g., the system or hardware scheduler) might be slowing our fellow worker down, and we could perhaps speed up the whole thing by taking it upon ourselves to accomplish the same task.

However, this is assuming we are able to assist other threads at all. It requires that multiple threads be allowed to work on a same transition simultaneously. This is far from the norm for most task-based environments, where each unit of work is usually exclusive and locks down several shared resources for use by the particular thread it is assigned to. Fortunately, in our case, the whole of last chapter was dedicted to building a non-blocking data structure to represent Kahn processes and channels, one which supports concurrent overlapping transitions.

Therefore, though we have taken a different path, we end up with a similar conclusion to the one we drew in the previous chapter about fair sequential schedulers: to attain lock freedom, it suffices that we can inject some fair selection (e.g., picking randomly from the A set) in the wait times of our blocking algorithm.

With these additions, the specification of our process scheduler—or at least an idealized version thereof—can thus be considered satisfactory, and we now turn our attention onto the implementation side. At a glance, the code features a lot of disjoint-set operations, all of which move an element from one set to the next. As we recall from our lock-free adventures in the past two chapters, moving things between shared data objects is usually difficult. Previously, we have solved the various cases that have appeared before us (e.g., pushing to and popping from macro queues) by using indirection backed by election-like algorithms (in the form of compare-and-swap) and assistance. In this chapter, however, we take a different route altogether.

Moving something from one place to another does not exist as a primitive operation; we only have assignments, and, by extension, copying. Thus, to move an item from one set to another, we need to combine two actions: deleting from the old and inserting into the new. The two major risks associated with this intervention are as follows.

- If we perform the statements in that order—delete before insert—then we need to store the content in some temporary (local) variable first. The risk is therefore that the driving thread halts and the moving value is **lost** forever, since it is absent from the first set, and not yet present in the second.
- If we perform the statements in the reverse order—insert before delete—then symmetrically there is the risk of **duplication**, if the thread stops before it removes from the old set.

In general, duplication is more acceptable than loss, since at least, there is still a chance we might realize and fix it in the future, whereas once a value is gone from the system, it cannot be recovered, by definition (else it would not have been really gone, merely hidden or scattered in some other form as redundant information somewhere). This is why compare-and-swap is eventually involved, at some point, to merge back different versions of a same entity, by requiring consensus, which lets threads know about each other and synchronize.

However, we should note that a single compare-and-swap instruction is able to unify possibly as many copies as we want. Therefore, a basic principle follows: there is no need to have such merge guards at every move operation. It suffices that eventually one happens. This is a boon for us, since our data structure that allows equivalent transitions to happen simultaneously already handles deduplication. Moreover, the process graph we thus maintain also always lists every possible process in existence.

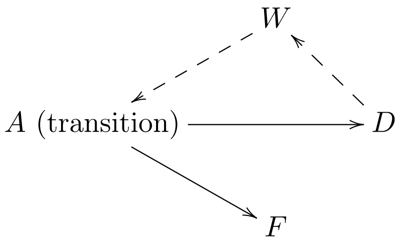
Therefore, there is no need for us to repeat what has been done in the previous chapter. More specifically, it would be possible to represent disjoint sets using some sort of marker on the processes themselves, while memory does not move at all—essentially using indirection, as we did in Section 3.7.2 for macro channels. Another possibility, as we mentioned in the same section, is to make use of some sort of full transaction, such as a multi-word compare-and-swap operation. However, keep in mind that even with the latter, moving protected pointers to blocks of memory obtained from a lock-free heap is even more expensive than it is for plain values. Indeed, read sections need to be established for each thread participating in assistance, lest the chunk risk being recycled while the transaction is still considered underway by some—similarly to how we handle joint updates to process-bound descriptors and channel blocks in Section 3.7.1.[3] As we have been saying, however, none of this is necessary in our case, since we can delegate all of the hard lock-free-related work to the contraptions of the last chapter.

Instead, we build our scheduling functions as if they were blocking, with a twist: we deliberately allow process identifiers to be duplicated between the work sets and the earlier authoritative Kahn process graph, which acts as a secondary scheduler, less efficient but exact. Therefore, although the algorithms are blocking, they are not permitted to wait explicitly without making progress. Instead, the *Wait* statement of Figure 4.3 is replaced by a special failure code that tells the caller to switch to the secondary scheduler, which draws tasks directly from the graph data object instead of the usual work sets.

In this design, the primary (blocking) and secondary (non-blocking) schedulers work completely separately. In particular, once *getTask* is aborted, we do not call *putTask*: the graph-based algorithm does not touch the work sets defined previously. For example, if a thread running a given process is delayed, another worker may fetch the same element by scanning the graph, but is not allowed to put it back into W, or move dormant items from D to W on its behalf. This avoids introducing duplicates in the circuit, and allows sets to be implemented by weaker data structures that do not guarantee uniqueness.

Following these guidelines, Section 4.1.2 explores a couple of techniques to manage the W set—and later the A set—concurrently and efficiently. Section 4.1.3 deal with the remaining D and F sets, and the methods to detect and synchronize information about

[3] Note that, when using a memory management system based on reference counting, it is possible to carry over the references within descriptors, which simplifies the equation quite a bit; with hazard pointers, however, it requires a non-trivial protocol, such as the one we used in Section 3.7.1.



Arrows represent moves from one work set to another. Delay-induced losses are not represented: lost elements end up in A implicitly. Dashed lines do not enforce uniqueness, while full lines (after a transition) do.

Figure 4.4: Scheduling cycle

completion, be it of transitions or whole processes.

This study is completed by the more open-ended Section 4.1.4, which deals with the topic of correctly meshing the two scheduling levels. We touch upon a few ideas that go beyond the strict separation into two layers. We discuss what would be the consequences of allowing the backup scheduler to manipulate the primary work sets directly, and how we might want to deal with them.

4.1.2 Dynamic load balancing

4.1.2.1 Focused process selection policy

The first thing we need to be wary of, when looking for an implementation of the W set, is the presence of caching of process states by worker threads, as we have demonstrated in the previous chapter. While they are very welcome for performance reasons, private process states must be taken into consideration when designing a scheduler. Indeed, the algorithm in Figure 4.3 dictates that processes having completed a transition be returned to W for further fair selection (line 7). However, this assumes that the state of the x process just executed has been shared; otherwise, there is no point in pushing into W as no other thread can take advantage of the progress just made.

Therefore, a more sensible caching-aware algorithm should instead **focus on a single process** as much as possible, rescheduling x again and again as long as it completes its transitions (thus does not need to wait) and does not terminate. In this way, the process rejoins the W set only when it either finishes or waits.

In both cases, its shared state should be updated. We remark that this is not the natural behavior of the core interpreter we have designed, which presently only synchronizes the shared state minimally—whenever it would be incorrect to do otherwise. Notably missing is the wait case, which does not trigger an update in the version shown in the last chapter. However, manually forcing the local state to be published is easily achieved by calling *setState* from *copoll*, when informed by *cocall* that a transition has been left incomplete. It is also transparent to existing routines, as the effects of *setState*

```
putTask($x$):
    $A \leftarrow A \setminus \{(T, x)\}$
    **If** $\exists u,\ [x$ has just completed a transition on $u]$
        **If** $\exists y,\ (y, u) \in D$
            $D \leftarrow D \setminus \{(y, u)\}$
            $W \leftarrow W \cup \{y\}$
        **return** $x$
    **Else if** ensureSharedState($x$)
        **If** $\exists u,\ [x$ is waiting on $u]$
            $D \leftarrow D \cup \{(x, u)\}$
        **Else if** $x$ is finished
            $F \leftarrow F \cup \{x\}$
        **Else**
            // Incomplete transition due to a macro operation.
            **return** $x$
```

Figure 4.5: Caching-aware blocking scheduler

are completely absorbed by the following invocation of *getState* that begins the next *cocall*.

This is our first hook into the existing code. The pseudo-code is given in Figure 4.5, where *ensureSharedState* makes sure, as its name indicates, that the local state has been pushed into the shared graph structure (either by a prior call to *cocall* or by invoking *setState* itself), and returns true if it succeeds, or false if another thread has beaten us to the update.

With this, each process goes through a scheduling cycle that alternates between W, A, D and F as illustrated by Figure 4.4.

4.1.2.2 A shared work set

As a starting point, let us note that the most direct implementation of the algorithm from Figure 4.3 casts W as a global shared collection. Any collection that exposes an insertion and fair selection operations can do the job. Perhaps the one that first comes to mind is the first-in first-out multi-producer multi-consumer queue. In this case, we replace random selection from a set with the round-robin behavior of a queue, which also ensures fairness and can be fairly efficient. Since we already have a lock-free memory management system, we could use, for example, the list-based non-blocking queue of Michael and Scott [1996].

But since we allow ourselves to use blocking algorithms, what prevents us from applying mutual exclusion to the entire W set? Instead of waiting for the lock, other threads would simply degrade to the fallback option of scanning the process graph, as explained above. While it is technically feasible, in the sense that it would still produce a correct algorithm, such an approach would be very sensitive to delays, as there is no way to

```
struct Deque {
    atomic int bottom;
    atomic int top;
    Array ∗ atomic array;
};

struct Array {
    int size;
    atomic void ∗buffer[];
};
```

Figure 4.6: Chase–Lev work-stealing deque: declarations

recover once the lock is lost to some delayed thread.

Compare this to the previous idea of plugging a stock lock-free queue to play the role of W. Doing so does not prevent tasks from temporarily disappearing when they exit the scope of the queue and become local for the duration of a transition, as explained in Section 4.1.1.3. However, even if a thread T halts in this way, the damage is limited: only one process is lost and needs to be handled by the secondary scheduler, which is not necessary until the rest of W is exhausted. This gives T time to rejoin the ranks, if at all possible.

In general, we therefore concentrate on almost lock-free algorithms—what is sometimes referred to informally as “lockless” in some circles—even on our blocking side. The objective is to isolate losses to single elements transiting between sets, in order to conserve the desirable damage-control quality just cited.

4.1.2.3 Work stealing

An arguably more interesting option is to distribute the contents of W among worker threads. One popular technique that achieves this is the work stealing of Blumofe and Leiserson [1999].

In work stealing, each thread owns a specialized work deque that contains tasks to be scheduled (references to processes, in our case). New items are only ever pushed to one’s own deque, but can be either be taken locally from the same structure, or stolen from a remote deque. Of course, threads give priority to the local source. The overall ensemble of deques acts as the single set W.

A lock-free version of this algorithm has been proposed by Chase and Lev [2005], although it depends on unbounded integers. The code is reproduced in Figures 4.6 to 4.9, in the form of three methods: *give*, *take* and *steal*.

The algorithm offers an asymmetrical deque: the owner thread pushes and pops elements from the bottom, as if it were a stack, while thieves steal from the top. In this way, there is contention only when the owner stumbles upon the a thief, i.e., when there is only one element left. At that point, compare-and-swap is used to decide a winner for

```
 1  bool give(Deque *q, void *x)
 2  {
 3      var b := q.bottom;
 4      var t := q.top;
 5      var a := q.array;
 6      bool ok;
 7      if ($b - t \leq a.size - 1$) {
 8          a.buffer[b % a.size] := x;
 9          q.bottom := $b + 1$;
10          ok := true;
11      } else {
12          // Full queue; can be resized by swapping in a new Array.
13          ok := false;
14      }
15      return ok;
16  }
```

Figure 4.7: Chase–Lev work-stealing deque: *give*

the last prize.

The basic insight is reminiscent of the single-producer single-consumer queue. Indices control what threads see or do not see of the underlying array. When one side is controlled entirely by a single thread, it can manipulate the index using inexpensive load and store instructions to dictate the boundaries of what is shared and what is not (hence only accessible to that thread). If, however, there is a risk of contention, compare-and-swap is used to resolve the situation.

Lemma 21. *The Chase–Lev work-stealing deque is linearizable.*

Proof. Following the above intuition, we give a brief and informal—more so than previous proofs—explanation of the linearizability of the work-stealing deque; for more details, please refer to [Chase and Lev, 2005].

We can see that *give* is linearizable, either upon increasing the index on line 9 (if successful) or upon reading the *top* index on line 4 (if not). Data is visible only when the index moves, and since only the current thread can add more items, if the deque was spacious enough before, it remains so now. Conversely, loading *top* is the only shared operation in a failed run, thus everything occurs seemingly instantly at that point.

The *take* method is more complex, arguably, the most complex of the three. It has three main paths:

- If we find an empty queue, then by the same reasoning as above, we linearize upon reading *top*, on line 6.
- Otherwise, there is at least one element when we first enter. If it is not the only one, compare-and-swap is not needed and linearization happens upon writing to

```
void ∗take(Deque ∗q)
{
    var b ≔ q.bottom − 1;
    var a ≔ q.array;
    q.bottom ≔ b;
    var t ≔ q.top;
    void ∗x;
    if (t ≤ b) {
        x ≔ a.buffer[b % a.size];
        if (t = b) {
            // Last element.
            if (not cas(&q.top, t, t + 1))
                x ≔ null;
            q.bottom ≔ b + 1;
        }
    } else {
        x ≔ null;
        q.bottom ≔ b + 1;
    }
    return x;
}
```

Figure 4.8: Chase–Lev work-stealing deque: *take*

```
void ∗steal(Deque ∗q)
{
again:
    var t ≔ q.top;
    var b ≔ q.bottom;
    void ∗x ≔ null;
    if (t < b) {
        var a ≔ q.array;
        x ≔ a.buffer[t % a.size];
        if (not cas(&q.top, t, t + 1))
            goto again;
    }
    return x;
}
```

Figure 4.9: Chase–Lev work-stealing deque: *steal*

bottom, on line 5, as *top* is guaranteed to be less than or equal to the value we have read in the actual invocation, on line 6, due to monotony. (Notice this is only true because we write to *bottom* before reading from *top*.)

- Else, we fight for the single last item. Whether we win or lose at compare-and-swap (line 12), we can linearize there. If we win, then any competing *steal* invocation fails and loops, totally discarding the results of previous iteration, and are therefore equivalent to fresh invocations (see linearization of *steal* below), started after our compare-and-swap on line 12. If we lose, then we could have read *top* later and gone directly on the empty path.

Finally, *steal* is the only method that contains a loop. As mentioned when discussing *take*, this makes it so failed iterations appear as if they did not happen and the invocation simply started later. There are, in fact, only two meaningful paths in *steal*: exiting with an element or null to signify emptiness. In the case of the latter, we linearize at the read from *bottom* on line 5. Since *top* monotonically increases, its value can be no less than on the previous load, on line 4, therefore also indicating an empty deque. If instead *steal* returns some item, then we can pick compare-and-swap, line 10, as the linearization point, knowing that any competing (failed) *steal* operation discard their iteration, thus linearize after that point, and any concurrent *take* occur at their own unsuccessful compare-and-swap (see above) in the linear history, therefore seeing the result of this invocation and bailing out with an empty result, as expected.

Note that the above argument is more subtle than it sounds, as there is in fact no point in the concurrent history where the *steal* invocation could take place atomically without affecting other invocations. We also need to rewrite any concurrent *take* instance into a call that goes through the direct path (no compare-and-swap), in order to get a valid linearization. Alternatively, instead of considering the later occurrence of *take* in the linear history, we could also linearize *steal* early, before line 6 in the competing *take*, thus similarly rewiring it into its direct path. □

With this, we can easily implement a W set for our scheduler. Since Kahn process networks do not follow the structured fork–join paradigm work stealing was originally designed for, none of the theoretical bounds apply. However, it still makes for an effective way, in practice, to spread the contention, especially with the Chase–Lev algorithm.

Work-stealing deques and integer overflow In order to be useful in an entirely lock-free system, however, we cannot rely on infinite integers. After all, if we had settled for that in the beginning, there is no doubt that much of the work so far would have looked vastly different.

The monotonic-block technique that we have used up to now still works: we wrap the whole object in a protected pointer that requires safe reading. After all, infinite indices are just another case of monotonic counters. The problem is made more complex by the fact that, similar to queues of queues, the current index interval can span two containers at once, when the producer reaches the end of the current range, but consumers are still

$\langle T_0 \rangle$ x pushes into $u \Rightarrow$ true
$\langle T_0 \rangle$ x pushes into $u \Rightarrow$ false
$\langle T_1 \rangle$ y pops from $u \Rightarrow$ true
$\langle T_1 \rangle$ y pops from $u \Rightarrow$ false
$\langle T_0 \rangle$ $D \leftarrow D \cup \{(x, u)\}$
$\langle T_1 \rangle$ $D \leftarrow D \cup \{(y, u)\}$

Figure 4.10: Non-injective mapping in D

lagging behind. It is even more complex in the case of a work-stealing deque, since the owner can waver back and forth with *take* and *give*, whereas our macro queues could only go one way.

One solution to this is to maintain two deques at once. Once *bottom* reaches the maximum, the owner thread switches to using *steal* on itself to remove the remaining elements and push *top* to saturation, at which point the object is swapped out. Meanwhile, the other deque is used to store new tasks introduced by *give*.

While this method is clearly quite simple, following the pattern we have seen several times by now, it suffers from one blatant issue: it does not follow deque semantics. Indeed, at the point when indices must wrap, the owner switches to self stealing, which does not follow the order of usual *take* operations. The object thus effectively morphs into some data structure that is almost a deque, but not quite.

However, as we have already stated, Kahn process networks do not benefit from this last-in first-out behavior in any theoretically significant way, except for the reduced contention. Therefore, while, to the best of our knowledge, it has not been used elsewhere, and may not prove so useful to others, it is quite sufficient for us.

4.1.3 Passive waiting

4.1.3.1 Waiting on channels

Next on the list is the D set. Unlike W, this one is very much specialized, as it is really a function from processes to channels. We might argue that it should be injective. After all, in a correct program, each channel is bound to one producer and one consumer: the first waits if there is no space, the second, if there is no data. The two options are mutually exclusive, since a channel cannot be both empty and full at the same time. However, due to concurrency, it is still possible for D to have both elements temporarily associated with the same channel, as shown in Figure 4.10.

The straightforward implementation of D is thus a pair of variables in the *Chan* structure, e.g., an additional *dormant* array indexed by the *IN* or *OUT* directions. Each cell is null when empty, and a process pointer otherwise.

Putting a process to sleep in one of the D cells is not as easy as writing a single value to memory, however. As we have just seen in Figure 4.10, there is a potential race between one operation and the opposite side of the channel. This example is actually a

```
   // Returns true if the current (cached) transition can be completed.
   bool canComplete(Proc *);

 1 // To be called after an incomplete transition.
 2 bool putToSleep(Proc *p, Chan *ch, ArgKind kind)
 3 {
 4     var ok := canComplete(p);
 5     if (not ok) {
 6         ch.dormant[kind] := p;
 7         var ok := canComplete(p);
 8         if (ok)
 9             ok := cas(&ch.dormant[kind], p, null);
10     }
11     return not ok;
12 }
```

Figure 4.11: Passive channel wait: *putToSleep*

specific case of a larger problem, which can be described as follows.

Suppose we attempt a transition on some process x, which fails to complete due to channel u. We then proceed to add (x, u) to D. However, before we are allowed to, y, the process on the other side of u, pushes to or pops from u, and immediately examines D for something to wake up. Yet, it finds nothing, since x has yet to be moved there. If y then itself blocks on some input or output (possibly on u itself, as in Figure 4.10), this can create a nasty dependency cycle.

Of course, as always, we could rely on the fallback graph scan to cover up for us. However, this would mean losing processes from our primary scheduler even though no delays or faults occur. In other words, the algorithm would be incorrect when used standalone, even in a blocking context, which is definitely not what we want. In keeping with the spirit of the above subsection, we should instead strive to limit the blocking behavior to those cases where tasks are held transitorily by threads during a move. In cases, like this one, where the items are still present in the shared work sets, we should not have to resort to waddling through the Kahn process graph.

The expected result can be achieved by double checking the feasibility of the transition after saving to D. Notice that in Figure 4.10, if T_0 looks back and reattempts to push into u at the end, it will go through.

The procedure *putToSleep*, in Figures 4.11 and 4.12, implements this idea. It puts its argument p to sleep on the given channel *ch*, then checks whether the last transition on p can be completed (line 7). If true, we attempt to take back the process (line 9), since it should not be asleep. The *putToSleep* function returns true to indicate that p has indeed been put to sleep, or false otherwise. If false, the caller still has ownership of p and should retry the transition.

```
1  // To be called after a complete transition.
2  Proc *awaken(Chan *ch, ArgKind kind)
3  {
4     var p := ch.dormant[kind];
5     if (p $\neq$ null) {
6        if (cas(&ch.dormant[kind], p, null))
7           return p;
8     }
9     return null;
10 }
```

Figure 4.12: Passive channel wait: *awaken*

Lemma 22. *In the context of a blocking scheduler, the* putToSleep *and* awaken *procedures are linearizable.*

Proof. We assume exclusive ownership of tasks, according to Figure 4.4. Therefore, on entry to *putToSleep*, the *dormant* cell must be null (otherwise, we would not be in possession of this process), and no concurrent thread can set it to a non-null value. The only competition permitted is in the form of *awaken*.

If *canComplete* returns false on line 7, then the call linearizes on line 6, as retrying earlier would also certainly lead to failure.

If *canComplete* returns true on line 7 and compare-and-swap succeeds, then we linearize there, on line 9. Indeed, the value has not changed since we set it to non-null, therefore, there has been no concurrent compare-and-swap from *awaken* between line 6 and line 9, thus we can safely move all the action to that point.

If *canComplete* returns true on line 7 and compare-and-swap fails—thus has no effect—then the invocation behaves as if we simply hand over the task without doing anything more. We thus linearize at the first check, on line 4. Indeed, at that point the winning compare-and-swap from *awaken* has yet to happen, and the state does not allow for further progress by virtue of *canComplete* returning false once. Therefore, the linear equivalent skips compare-and-swap completely and simply returns true.

For *awaken*, the procedure takes effect on line 4 if that test fails, since it is then the only shared access.

If *p* is non-null, then *awaken* linearizes at compare-and-swap. If it is lost, then *dormant* is empty at that point and the linear invocation returns null by reading null directly from the field the first time on line 4. If it is a win, then we are first on compare-and-swap and at that point, the value is sure to be non-null, therefore the whole invocation could very well take place instantly right there. □

Corollary 8. *If every complete transition is eventually followed by a call to* awaken *on the opposite side of its channel, then there is no* (x, u) *in* D *such that* x *can make progress.*

Proof. By Lemma 22, we consider an equivalent linear history containing instances of *putToSleep* and *awaken*. Suppose there is a process x in D that can make progress. Then there must be an invocation of *putToSleep* that adds x to D and a complete transition that allows x to continue. By hypothesis, that complete transition is followed by a call to *awaken*. If *putToSleep* happens before *awaken* then the latter removes x from D. Else, *putToSleep* happens after, the call to *canComplete* returns true, and therefore x is not added to D. Impossible. □

Corollary 8 states an important result: as long as *awaken* is called sufficiently often, no process is wrongfully blocked in D when it could be placed in W for immediate execution. Furthermore, "sufficiently often" does not mean after every transition; it only needs to happen eventually.

This leads to a number of **lazy waiting** strategies, regarding when and how often we check for processes to move out of D. The most straightforward application is to buffer up on required *awaken* calls, and batch them together once the list is full. This allows multiple operations on a same channel to be grouped, necessitating a single *awaken*. Needless to say, however, that this is a very blocking tactic, so we should be careful not to oversize our wait buffers.

An alternative is to call *awaken* only when the worker thread releases ownership of the process; it then scans and awakens every adjacent node in the graph. These invocations cover for every possible transitions that have been performed by the thread on behalf of the process, when it was selected. Incidentally, due to the focus policy, this makes it so *putToSleep* is preceded by a round of *awaken*.

Sleeping in the presence of a secondary scheduler Interestingly, these lazy waiting strategies are also needed in order to integrate with the secondary scheduler. In the presence of fallback scheduling, processes might progress concurrently, which only affects our two functions in so far as *putToSleep* calls *canComplete*. What do we do if not only the transition can be made, but has actually already been made? Certainly, the task should not be put to sleep, since the transition that required this to complete is already over. Therefore, it would seem that neither the code nor the proofs changes.

However, the bigger issue lies with the fact that *awaken* cannot be called by the secondary scheduler, by construction. How do we guarantee the alternation with complete transitions needed by Corollary 8?

The basic insight is that we can turn all **deadlocks into livelocks**, by rescheduling waiting tasks unconditionally, i.e., removing them from D, when out of work. As a side effect, this also eliminates any errors we might introduce ourselves due to the superposition of the two scheduling layers.

For instance, one solution is to use the awaken-before-sleep strategy described above. This works because, in this strategy, if even one item is left in W (or executing on some thread), then the whole connected component will eventually be woken up by breadth-first traversal, regardless of alternation or anything else. In effect, it replaces every possible deadlock by a corresponding livelock.

Another similar solution is to impose a full scan of the D set by every thread in *getTask*, before terminating. If every element is in F, then we are effectively done; otherwise, we loop on anything in D, thus again turning any deadlock into a livelock. Since we are talking about primary scheduling, this is a blocking action that removes elements from D to examine them one by one. If any of them is erroneously asleep, it is thus recovered, and the scheduler stops its termination procedure to resume normal operations.

4.1.3.2 Waiting for termination

This brings us to the second kind of passive waiting: termination, the F set. This one is even more specialized than D since elements moved to F never go back to any other work set. There is also little opportunities for unrecoverable errors, as terminated processes are not needed for progress by anything else in the network. Besides, once processes reach their final state, they are no longer updated.

The specificity here is different. As a matter of fact, detecting termination in the primary scheduler, which is sensitive to delays, is unreliable, by definition. Furthermore, we remember that we already have *checkFinished* routine from the previous chapter, that can play this role for the secondary scheduler. Therefore, the primary implementation of F serves a slightly different purpose: it provides hints as to when *checkFinished* should be called.

Quite simply, we keep a counter of definitely finished children in each process, that is updated only by the primary scheduler, therefore eliminating any possibility of duplicate counting. When this counter equals the total number of children, *checkFinished* can be called to join all the subprocesses.

In parallel, the secondary scheduler keeps its slow but reliable counting strategy, so that any missed children in the first layer can eventually be accounted for in the second.

4.1.4 With or without the secondary scheduler

In the previous subsections, we have been through several techniques that together build a credible blocking primary process scheduler. We now take a step back to look at the bigger picture: how our proposed implementation would fare both as a standalone scheduler, and in a non-blocking system, backed by a fair and reliable (and less efficient) deputy.

4.1.4.1 As a primary scheduler

We have already explained the basics of how the two scheduling layers should intereact in Section 4.1.1. A worker always refers to the primary, and only queries the secondary if the first returns null. This can signify both network termination, if the secondary scheduler confirms, or a blocking condition, in which case it provides a fallback option.

In our base design, the second layer is not allowed to touch any of the data structures of the first. Therefore, interactions are limited interference at the level of process states and progress, as described in Section 4.1.3.1, and cooperation with regard to detecting termination, as seen in Section 4.1.3.2.

It would be tempting to allow more than this. Why not let the secondary scheduler push into W or D? For one, it seems like it could simplify the *awaken* alternation problem Section 4.1.3.1. Indeed, as we recall, the main issue was that transitions triggered from the secondary layer were not systematically followed by *awaken*. However, this does not account for the fact that under such a scenario, Lemma 22 and Corollary 8 do not hold, or at least not without some major algorithmic changes and code modifications, so at least the first store instruction is replaced by compare-and-swap, to handle concurrency (and potential delayed invocations) on the D set, which only allows one mapping for each process, as we have implemented it. Otherwise, we need to accept multiple pairs per process in D.

More generally, the problem is one of duplication: allowing the secondary level to modify the work sets directly kills the exclusivity that comes with blocking code. This, in turn, could lead to increased redundancy and contention, potentially infinitely so, if no measure is taken to control the inflation.

One radical way to deal with the situation is to put a user-defined limit to the size of the work sets, perhaps proportional to the number of processes it should contain, without duplicates. When the cap is reached, items—either new or old—are simply dropped, thus relying on the secondary scheduler to recover them, if they were not duplicates. This works in theory; however, it could suffer from high variance in its performance, depending on how much garbage is present in the work sets. Moreover, the algorithms in their present state are made to conserve every element. Most likely, this strategy would require the addition of some form of duplication control heuristic to complement the safe nature of the construction. Whether all those additions would be worth paying for is another question, as with each of them, the difference in performance between the two layers decreases.

In contrast, our suggested build, which is far simpler, also exhibits almost unhindered performances when running with little delays. We have been careful in our choice of algorithms to bound the number of temporarily lost tasks to a small constant—basically only those being run or transiting from one set to another. As noted before, this gives ample time for delayed threads to recover. Morever, the strict separation between the two scheduling layers might be a drawback if workers start halting for long periods of time, but it is also an advantage when dealing with short pauses. Indeed, the impact of a few stray secondary picks are unlikely to significantly affect the course of the main scheduling and execution effort.

This can be enhanced by making the non-blocking scheduler smarter, and allowing it to provide hints to the first layer. Although we want to guarantee maximal progress in the fallback selection mechanism, it does not completely rule out cleverness: we can have weighted choices, and intelligent graph traversal. For example, we can prioritize processes that do not seem to be in D, or concentrate on finding processes that seem stuck (not in W either). As for hints, we could imagine a non-authoritative subset of D, updated by the secondary to inform the primary of elements it should look at before embarking onto a full scan of D to find missed awakenings (as explained at the end of Section 4.1.3).

Lastly, although we believe it may be unnecessary, there is nothing preventing us from designing an efficient non-blocking backup scheduler in itself. As an example, just using local counters to keep track of what processes draw more contention on their shared states, it is possible for threads to assign lower priorities to those tasks that are likely already taken by other workers, and attain some form of dynamic partition. Add to that some form of demand-driven traversal, and we should have decent locality.

We believe this shows our design to be a reasonable compromise for a general-purpose Kahn process implementation, with the potential for great peak performance under ideal conditions, and reasonable guaranteed behavior in all situations.

4.1.4.2 As part of a blocking Kahn process implementation

We remark that, if used in a fault-intolerant environment, the blocking algorithms described in this section make a perfectly fine scheduler by itself. As we have noted several times, it has always been our goal not to rely too much on the presence of a secondary layer, so that the primary one could act as a standalone entity.

If we go down this route, there is no need to burden ourselves with the complexity of our non-blocking graph implementation. We can basically keep the *Proc* tree as it is, and simply swap in a more efficient single-producer single-consumer channel implementation, such as the one we present later in Section 4.2.2. Since our design grants the executing worker thread exclusive access to the running process, the single-producer single-consumer contract is satisfied: at any one time, only one thread sits on each end of the channel.

This leads to a very efficient blocking Kahn run-time library, which we have actually implemented in C11 and experimented on, as a benchmark for future implementation work on a fully fault-tolerant system—the non-blocking core components have yet to be realized in C11 at the time of writing. The blocking version serves as the basis for the performance demonstrations of section Section 4.3, which show how, at a higher level, Kahn processes can be used to match, or in some cases surpass, other task-based parallel paradigms currently in use.

A key factor in achieving such results—and that we believe makes our lock-free efforts promising—is the perspectives of low overhead of our algorithms when ported to a relaxed memory setting. This is the focus of the next section, as we turn our attention to further implementation challenges and more performance-oriented discussions.

4.2 In relaxed memory

In this section, we discuss the difficult problem of concurrent algorithms when faced with the harsh reality of weak memory properties. In contrast to the sequentially consistent view we have held up to this point, the world of both hardware and language concurrency consists mostly of what are termed **relaxed memory models** (or weak memory models), which are, broadly speaking, anything that guarantees less than sequential consistency.

In a weak model, at the lowest level, execution is not made of interleaving instructions. Instead, unordered statements can exist, neither acknowledging the effect of the other.

Imagine two threads. One sets x then reads y; the other one does the opposite. Surely, at the end of it all, at least one of them should see x or y set, depending on which of the store instructions happens first. That is certainly the case under sequential consistency, and is the basis for the anti-store-buffering pattern, as seen previously in Sections 2.2.3.3 and 2.6.4. However, this may or may not be the case under relaxed rules: it could very well be that both x and y end up being loaded as unset, if the actions of the two threads are considered unordered.

To work around this, additional primitives are offered, that control what must be related by some order and what can be left floating. These usually take one of two forms:

- variations of the basic operations, such as load, store, and compare-and-swap;
- or special **fence** instructions to be inserted between other statements to force certain constraints.

Some popular contemporary memory models are C11 (the memory model of the C language in its 2011 revision, as formalized by Batty et al. [2011]), x86-TSO, POWER and ARM, all of which have their own quirks and generally lead to subtly (and less subtly) different architecture-specific code patterns. All of our implementation work utilizes the C11 language. Therefore, although we try to introduce new ideas and arguments in a more neutral way, whenever possible, the proofs themselves, presented in this section and the corresponding appendices, all depend on the C11 memory model.

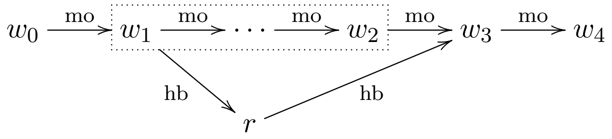
It is generally admitted that every non-esoteric concurrent platform, including C11, must be able to somehow provide basic linearizable objects and routines, such as load–store registers and compare-and-swap—although they may not be available directly as primitives and may not play well with the rest of the field. As such, at the very least, all of the algorithms presented in the previous pages could be used without change on virtually any realistic system that would need them. They would run. However, without relaxation, they would also certainly suffer some performance penalty. Hence, the interest in the kind of low-level optimizations we discuss in this section.

Sections 4.2.2 and 4.2.3 provide in-depth commentary of two major components, single-producer single-consumer channels and work-stealing deques, with proofs of ad hoc important properties. In Section 4.2.4, we briefly review more relaxation opportunities. Finally in Section 4.2.5, we talk about the disparities between C11 and the standard linearizable world we come from. We discuss how the new relaxed optimizations could potentially be integrated into the existing linearizable blocking or non-blocking interpreter, and how, conversely, our run-time library could be called from user code written in C11, in the future.

Before we start with detailed coverage, however, let us first quickly introduce the C11 memory model, from a programmer's point of view.

4.2.1 The C11 memory model in two useful patterns

C11 distinguishes between **atomic** (denoted by the *atomic* qualifier we have seen in many codes), which may be accessed concurrently, and non-atomic objects, which may



Each instance of w denotes a write action, while r is a lone read action. Together, write actions are sorted by modification order (*mo* arrows). The dotted box represents all the write statements r may potentially read from, given the happens-before (*hb* arrows) relations shown.

Figure 4.13: C11 coherent reads

not. The rules we are about to present only apply to the former, and it is the job of the programmer to ensure that writes to non-atomic variables are exclusive with any other operation on the same location, a pact known as **data-race freedom**. That being said, let us concentrate on atomic objects.

Informally, executions of a C11 program are sets of actions (e.g., read, compare-and-swap) linked by binary relations, most of which are partial orders, themselves constrained by rules. Together, they define what can or cannot occur, when running the program.

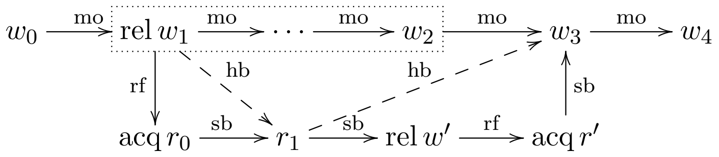
Most importantly, those relations regulate what values are read from and written to shared variables. Each shared location acts as a (potentially infinite) list of values, provided by operations that write to the location. Those operations each store a new value at the end of the list; the order thus induced between writes is called the **modification order**. Each statement that requests a value from some shared location is matched with a unique write operation in a relation known, quite intuitively, as **reads-from**. In the absence of further constraints, threads can read any value from any variable. Most of the additional constructs in the memory model serve to restrict that freedom.

4.2.1.1 Message passing

In sequential consistency, load and store instructions are all ordered with respect to each other, and an action always reads from the most recent write to the same location. In C11, things are, as expected, more relaxed. The main mechanism that limits what can or cannot be read at some point is the **happens-before** relation. Although the terminology is similar, happens-before in the realm of C11 is not a total order, as in sequential consistency. However, it is still compatible with both the modification and reads-from orders seen above.

In a sense, two events not related by happens-before can be considered truly concurrent. Basically, the behavior is relaxed in that there are now multiple candidates for each read operation: in addition to the most recent write that happens before, it can also read from any instruction that is not known to happen after, as shown in Figure 4.13.

The objective in concurrent algorithms is generally to bound the range of values that can be read in each position between two well-known modification statements. This can be achieved by sandwiching the operation between happens-before edges, as shown in



As above, w and r denote operations to a same location; w' and r' to another location. Arrows mark relations and carry the initials of the order they represent, i.e., *rf* for reads-from, *sb* for sequenced-before. The *rel* and *acq* superscripts indicate release and acquire operations, respectively.

Figure 4.14: C11 release–acquire synchronization example

Figure 4.13.

The most basic way to ensure that an action happens before another is to place them in a same thread. This is called **sequenced-before** in C11 and is simply the sequential thread order, which is part of happens-before.

Between threads, arguably the main method to restrict possible values available for reading is the **message-passing pattern**, where a reads-from pair is promoted to happens-before through the use of **release–acquire** synchronization. Formally, a reads-from relation between a release-qualified write instruction (annotated with *rel* in code and figures) and an acquire-qualified read instruction (*acq* in code and figures) is promoted to happens-before.[4] Intuitively, release–acquire is a propagation mechanism: a releasing thread publishes all of its knowledge at the time of writing to a potential acquiring reader. In hardware terms, on current machines, this forces the sender to commit any prior write operations, and the reader to postpone any subsequent reads to after the acquire instruction has completed, thus ensuring that any modifications preceding the release are visible after the acquisition.

C11 provides specific variations of write primitives that have release semantics; similarly for read primitives with acquire semantics. The synchronization occurs if an acquire operation reads from a release write, as in Figure 4.14. Observe that the reads-from edge from w_1 to r_0 links a release write to an acquire read, and is therefore promoted to happens-before. The same happens between w' and r'. This results in the dashed happens-before relations, that occur as the union of the underlying sequenced-before and (promoted) happens-before edges.

A similar effect can be accomplished by placing fences: a release fence before the write, and an acquire fence before the read, as shown in Figure 4.15. In this case, the synchronization is between the fence instructions, thus moving the targets of the happens-before relation. Most importantly, this feature can be used conditionally. This is especially useful on the acquire side, when polling or double checking for a specific value, for example.

A special use case of release–acquire synchronization is to have multi-word objects updated using completely relaxed instructions, followed by a single release write to a flag to indicate that the previous data is ready to be read. Hence, message passing.

[4] Formally, it is part of the synchronizes-with relation, which is part of happens-before.

$$\begin{tikzcd}
\cdots \arrow[r, "\mathrm{sb}"] & \mathrm{rel}\, f \arrow[r, "\mathrm{sb}"] \arrow[drr, "\mathrm{hb}"] & w \arrow[d, "\mathrm{rf}"] & & \\
& & r \arrow[r, "\mathrm{sb}"'] & \mathrm{acq}\, f \arrow[r, "\mathrm{sb}"'] & \cdots
\end{tikzcd}$$

Similarly to how w and r represent write and read operations, f denotes fences.

Figure 4.15: C11 release–acquire fences

4.2.1.2 Total ordering

The message-passing pattern creates happens-before edges by chaining alternating write and read instructions. Therefore, constraining a load statement can only be done relative to a preceding read and a following write operations. When used alone, release–acquire can be used to build single-producer single-consumer queues, as we will see in Section 4.2.2. In conjunction with compare-and-swap, many more small objects are possible. However, many times, we need stronger ordering.

This is where the next instruction set comes into play. The C11 memory model offers its own set of "sequentially consistent" primitives (denoted by the *sc* qualifier, in code and figures) that can be used to write linearizable algorithms on top of C11. The standard makes it clear that if a program uses only those instructions to access shared memory, then it behaves as if it ran in a sequentially consistent environment.

Fundamentally, C11 guarantees that all instructions marked **sequentially consistent** are related, according to some separate partial order that never applies to any other kind of instructions. Hereafter, we refer to this order as the **sequentially consistent total order**, or sequentially consistent order, for short. It is implied by the happens-before and modification orders, but the opposite is generally not true.

When not used entirely by themselves, these instructions interact with the rest of the C11 kit in two important ways. First, they imply release–acquire, hence happens-before between a write and following reads to the same location.

Second is the existence of sequentially consistent fences. In addition to acting as both release and acquire fences at the same time, they also serve a role similar to (but weaker than) happens-before with regard to constraining what values can be read by a given statement. The rule is as follows: an operation r preceded (in sequential code) by such a fence f may read from the most recent visible write, or any later modification. The **most recent visible**[5] write w is the last one in modification order that satisfies one of the following configurations:

$$w \xrightarrow{\mathrm{hb}} r \qquad \mathrm{sc}\, w \xrightarrow{\mathrm{sc}} f \xrightarrow{\mathrm{sb}} r \qquad w \xrightarrow{\mathrm{sb}} \mathrm{sc}\, f' \xrightarrow{\mathrm{sc}} \mathrm{sc}\, f \xrightarrow{\mathrm{sb}} r$$

[5] The term might sound somewhat confusing; it actually designates the "most recent" modification that we can be sure is visible—it does not mean that others are somehow invisible. Moreover, technically, the C11 standard defines it only for happens-before and not for the sequentially consistent total order. For the sake of simplicity, we regroup the various cases in our explanation.

$$\begin{tikzcd}
w_0 \arrow[r, "\mathrm{mo}"] & w_1 \arrow[r, "\mathrm{mo}"] \arrow[d, "\mathrm{sb}"'] \arrow[rrd, dashed] & w_2 \arrow[r, "\mathrm{mo}"] & \cdots \\
& \mathrm{sc}\, f \arrow[r, "\mathrm{sc}"] & \mathrm{sc}\, f' \arrow[r, "\mathrm{sb}"] & r
\end{tikzcd}$$

Sequentially consistent operations and order are denoted by *sc*. Again, the dotted frame delimits those values that can be read by r. The presence of the fences prevents r from reading from writes ealier than w_1. The dashed arrow represents the equivalent happens-before edge that would be needed to enforce the same constraint.

Figure 4.16: C11 sequentially consistent fences

$$\begin{tikzcd}
w \arrow[r, "\mathrm{sb}"] \arrow[rrd, dashed] & \mathrm{sc}\, f \arrow[r, "\mathrm{sb}"] \arrow[d, "\mathrm{sc}"'] & r' \\
w' \arrow[r, "\mathrm{sb}"] & \mathrm{sc}\, f' \arrow[r, "\mathrm{sb}"] & r
\end{tikzcd}
\qquad
\begin{tikzcd}
w \arrow[r, "\mathrm{sb}"] & \mathrm{sc}\, f \arrow[r, "\mathrm{sb}"] & r' \\
w' \arrow[r, "\mathrm{sb}"] \arrow[rru, dashed] & \mathrm{sc}\, f' \arrow[r, "\mathrm{sb}"] \arrow[u, "\mathrm{sc}"'] & r
\end{tikzcd}$$

Depending on the ordering of f and f', at least one of the two dashed arrows correctly specifies a lower bound for the corresponding read operation, in modification order.

Figure 4.17: C11 anti-store-buffering pattern

A more detailed example is provided in Figure 4.16. As can be seen, these fences only provide half the benefits of happens-before, as the set of readable values is unbounded on the right-hand side. However, this is sufficient to write the very useful anti-store-buffering pattern, as demonstrated in Figure 4.17.

As such, there is some limited interoperability between the sequentially consistent subset of C11 and the rest of its more relaxed arsenal. Furthermore, through the use of fences, a pattern can be created that prevents store buffering.

4.2.1.3 Summary and other C11 features

In our experience, few constructs are actually required to efficiently implement most the algorithms we need. In this work, we limit ourselves to a subset of the concurrent language with only the following:

- sequentially consistent fences;
- sequentially consistent or release–acquire compare-and-swap;
- release–acquire load and store;
- relaxed (but still concurrent) load and store when used either to write identical values or behind a release–acquire pair;
- and non-concurrent (non-atomic) accesses to shared memory, which are guaranteed to be totally ordered by happens-before by the rest of the code.

We have already explained release–acquire operations as well as the sequentially consistent fences. Compare-and-swap behaves as one would expect: it reads and conditionally writes the next value in the modification order of some variable. It is thus atomic with respect to the modification order. The fact that we use the sequentially consistent variant means that all instances of compare-and-swap—targeting the same or different locations—are also related to respect to each other in the sequentially consistent total order.

Relaxed accesses basically have no inherent restrictions tied to them, except they must honor the sequential semantics, when executing in a same thread. We use the patterns presented in Sections 4.2.1.1 and 4.2.1.2 to constrain what can be read and when modifications are made visible by such relaxed operations.

In C11, relaxed instructions have bizarre and complex semantics. In particular, they permit self-satisfaction cycles, meaning an assignment may be conditional upon a test that is only possible if the same assignment indeed occurs. The reasons behind and the full range of implications of such a thing are far beyond the scope of our work. However, they are a must in many algorithms, due to their ability to conditionally convert into more powerful variants through the help of fences. For more information on the full extent of their semantics, see [Vafeiadis et al., 2015].

4.2.2 Case study: a relaxed single-producer single-consumer queue

Part of this subsection is reproduced from [Lê et al., 2013]. For more information, please refer to that paper.

The first component we study is a lock-free single-producer single-consumer queue. Although the channel methods are lock-free, they impose that actions on each side, consumer or producer, be totally ordered (by happens-before), which requires some kind of exclusion mechanism in the client, which is not lock-free by design. It is, however, ideally suited to a high-performance blocking Kahn process implementation.

We start from the simple Lamport queue presented before and add two features: batching and index caching. The first is the ability to group multiple contiguous operations on either side under a single index update. The latter permits lazy loading of the opposite index. The code is shown in Figure 4.18.

The producer never writes to *front*, nor does the consumer ever write to *back*, thus, following the C11 memory model, the loads to *back* (line 4 in *push*) and *front* (line 4 in *pop*) in their exclusive method do not require restrictive memory orderings.

The release and acquire qualifiers on loading the opposite index and updating our own are rather intuitive uses of message passing. We want the data written on line 11 to be visible to the consumer after loading the index. Conversely, any reading on line 11 should be secured before telling the producer to overwrite the same cells. In C11 terms, this ensures data-race-freedom on the accesses to *data*, meaning it can be manipulated just like a regular variable, since all related operations are totally ordered by happens-before.

As far as C11-specific memory ordering goes, this is it. The rest is classic concurrent algorithms. Index caching, borrowed from the ring buffer of Lee et al. [2009], stems from the simple observation that each side only moves forward: the consumer does not

```
atomic size_t front;
size_t pfront;
atomic size_t back;
size_t cback;

// L must divide SIZE_MAX.
int data[L];

bool push(const int xs[*])
{
    var n := countof xs;
    var b $:=_{\mathrm{rlx}}$ back;
    if (pfront + L − b < n) {
        pfront $:=_{\mathrm{acq}}$ front;
        if (pfront + L − b < n)
            return false;
    }
    for (var i := 0; i < n; ++i)
        data[(b + i) % L] := xs[i];
    back $:=_{\mathrm{rel}}$ b + n;
    return true;
}

bool pop(int xs[*])
{
    var n := countof xs;
    var f $:=_{\mathrm{rlx}}$ front;
    if (cback − f < n) {
        cback $:=_{\mathrm{acq}}$ back;
        if (cback − f < n)
            return false;
    }
    for (var i := 0; i < n; ++i)
        xs[i] := data[(f + i) % L];
    front $:=_{\mathrm{rel}}$ f + n;
    return true;
}
```

Figure 4.18: Caching single-producer single-consumer queue

unconsume, the producer does not unproduce. It is implemented by reloading the opposite index only if we have caught up with the previously read value. Indeed, if we have been promised five values (or apples, or something) and only ate three and only we are allowed to eat them, then it is self-evident that the remaining two have gone nowhere and are still waiting at the same place.

Batching is another simple idiom. In the case where data can be aggregated, it makes sense to only update the index once per batch. Of course, this should not be made mandatory, as that would make the queue unsuitable for use as a generic channel in Kahn process networks—for the same reason we could not wait for whole monotonic chunks to fill before swapping them in the previous chapter.

We now move onto the crux of the matter: the proof of correctness in the C11 memory model. We prove two ad hoc properties. The first relates to call and return values, and states that successful *push* and *pop* invocations behave as if they were writing to and reading from the same infinite stream of values, value by value, in the same order. The second lifts the message-passing pattern to the granularity of the queue object: successful *pop* invocations return after the corresponding call to *push* (that writes the value consumed).

4.2.2.1 Preliminary definitions for the C11-correct queue

Actions and values For convenience, we represent initializations as pseudo-store actions. We note w_i^{f} (resp. w_i^{b}) the write action at position i in the modification order of variable *front* (resp. *back*); index 0 is the initial value and is assumed to be zero.

Other actions are denoted by their respective letter, and the appropriate superscript as needed, e.g., r^{f}.

Method invocations Since invocations on each side are totally ordered, let P (resp. C) be the sequence of push (resp. pop) operations. We note (P, k) the push of rank k and (C, k) the pop of rank k (counting from zero). The sequence of push (resp. pop) operations alternate between cached, successful uncached and failed push (resp. pop) instances. Those may appear as subscripts to denote actions happening within them.

- A **cached** push (resp. pop) determines locally that it has enough space ahead in the buffer, and does not reload the shared variable *front* (resp. *back*).
- A successful **uncached** push (resp. pop) observes that it does not have sufficient space left over from its previous operation to complete its current request, and reloads the shared variable *front* (resp. *back*). It then ascertains that sufficient space is available and proceeds successfully.
- A **failed** push (resp. pop) is an uncached push (resp. pop) that observes an insufficient amount of space available after reloading *front* (resp. *back*).

We note $(T, k)x$ (for $x \in \{f, b\}$ and $T \in \{P, C\}$) the cached value of x as can be observed sequentially at the beginning of the operation (T, k).

Thread-private variable are initialized to zero. Hence the first push operation in the producer thread will be uncached, and the first pop operation in the consumer thread will be uncached or failed. We subsequently define, for $T \in \{P, C\}$, the following functions on ranks of push and pop:

$$\llfloor k \rrfloor_T = \max\{i \leq k \mid [(T, i) \text{ is uncached}]\}$$
$$\llceil k \rrceil_T = \max\{i \mid \llfloor i \rrfloor_T \leq \llfloor k \rrfloor_T\}$$

Intuitively, $\llfloor k \rrfloor_T$ is the index of the nearest preceding uncached instance; $\llceil k \rrceil_T$ is the highest rank in the sequence of cached instances to which (T, k) belongs. We extend the notations $\llfloor T, k \rrfloor$ and $\llceil T, k \rrceil$ to mean $(T, \llfloor k \rrfloor_T)$ and $(T, \llceil k \rrceil_T)$, respectively.

Wrap-around and modulo arithmetic The shared variables *front* and *back* are implemented as $(\log_2 M)$-bit unsigned integers, where $M = \text{SIZE\_MAX} + 1$ in the C11 code. However, we treat the sequences of values written by w^{f} and w^{b} as not wrapped, in the proof; instead, the bit width constraint is reflected through the use of modulo-M arithmetic on the variables. This adjustment makes for easy distinction between equal wrapped values obtained from successive increments of *front* and *back*, but is otherwise equivalent. The $x \,\%\, y$ operation denotes the integer remainder of x divided by y, as in C. We note $w^{[i]}$ and $r^{[i]}$ actions on the memory location with index $i \,\%\, L$ in the underlying array backing the queue, where m is the size of the array. For this definition to match the C11 code given in Figure 4.18, L must divide M, so that $\forall i \in \mathbf{N},\ (i \,\%\, M) \,\%\, m = i \,\%\, m$. Additionally, if $M \neq \text{SIZE\_MAX}+1$, the remainder operations need to be made explicit.

4.2.2.2 Action structures of the C11 queue

We now formally define the three kinds of push and pop instances (cached, uncached and failed), previously introduced, and matching action structures through the control flow graph of the corresponding function. Figures 4.19 and 4.20 show all three action structures of a push or pop operation. Paths are split into their constituent shared-memory accesses, both atomic and non-atomic. For accesses to the data buffer, which depend on the batch size n (equal to the size of the *xs* array argument), only the first and last are represented. When multiple outgoing edges are possible, each one is annotated with the corresponding predicate condition under which it is taken.

4.2.2.3 Proof of the C11 queue

The proof is split in two. In the first half, up to Corollary 10, we prove useful invariants on the index variables *front* and *back*, using coherency and release–acquire semantics. These first results focus on establishing bounds on the locally observable values of the indexes (e.g., $f(P, k) \leq b(P, k)$ between the locally observable values of *front* and *back* in (P, k)). The latter half, from Lemma 29 onward, exploits these invariants to prove our three main theorems:

$$\begin{tikzcd}
\text{ini/rel}\, w_i^{\text{b}} \arrow[r, "\text{rf}"] & \text{rlx}\, r^{\text{b}} = b \arrow[r, "\neg\text{[cached } (P{,}k)\text{]}"] \arrow[d, "\text{[cached } (P{,}k)\text{]}"'] & \text{acq}\, r^{\text{f}} = f \arrow[rr, "\text{[failed } (P{,}k)\text{]}"] \arrow[dl, bend right, "\text{[uncached } (P{,}k)\text{]}"] & & \text{return false} \\
 & w^{[b]} \arrow[r] & w^{[b+n-1]} \arrow[r] & \text{rel}\, w_{i+1}^{\text{b}} = b + n \arrow[r] & \text{return true}
\end{tikzcd}$$

$$\begin{aligned}
[\text{cached } (P,k)] &= ((w^{\text{f}}_{(P,k)} + m - b) \,\%\, M) \geq n \\
[\text{uncached } (P,k)] &= ((f + m - b) \,\%\, M) \geq n \\
[\text{failed } (P,k)] &= \neg[\text{uncached } (P,k)]
\end{aligned}$$

Figure 4.19: C11 action structures of the *push* function

$$\begin{tikzcd}
\text{ini/rel}\, w_i^{\text{f}} \arrow[r, "\text{rf}"] & \text{rlx}\, r^{\text{f}} = f \arrow[r, "\neg\text{[cached } (C{,}k)\text{]}"] \arrow[d, "\text{[cached } (C{,}k)\text{]}"'] & \text{acq}\, r^{\text{b}} = b \arrow[rr, "\text{[failed } (C{,}k)\text{]}"] \arrow[dl, bend right, "\text{[uncached } (C{,}k)\text{]}"] & & \text{return false} \\
 & r^{[b]} \arrow[r] & r^{[b+n-1]} \arrow[r] & \text{rel}\, w_{i+1}^{\text{f}} = f + n \arrow[r] & \text{return true}
\end{tikzcd}$$

$$\begin{aligned}
[\text{cached } (C,k)] &= ((w^{\text{b}}_{(C,k)} - f) \,\%\, M) \geq n \\
[\text{uncached } (C,k)] &= ((b - f) \,\%\, M) \geq n \\
[\text{failed } (C,k)] &= \neg[\text{uncached } (C,k)]
\end{aligned}$$

Figure 4.20: C11 action structures of the *pop* function

- Theorem 3 establishes that calls to pop either fail or return a different element each time.
- Theorem 4 establishes that successful calls to pop read all elements pushed in the queue, without skipping.
- Finally, Theorem 5 asserts that the algorithm does not contain any data race, despite accessing its data buffer non-atomically.

The flow of this proof can be understood as building systems of inequations that substantiate the goals. We use local hypotheses (i.e., predicates listed in Figures 4.19 and 4.20) as elementary inequalities, further refined through constraints derived from the partial orders that comprise the memory model, as previously illustrated in Section 4.2.1.

Additionally, where needed, more complex constraints that expand significantly—and perhaps unintuitively—upon the premises of the memory model, are elaborated inductively. The inequations themselves consist of straightforward modular arithmetic. We included the calculations for completeness, but they should not distract from the construction of the formulas as the main contribution of this proof.

Theorem 3. *A value stored in the data buffer is never read more than once.*

Formally, given a store $w^{[i]}$ in an instance of push, there exists at most one load $r^{[j]}$ from an instance of pop, such that $w^{[i]} \xrightarrow{\text{rf}} r^{[j]}$.

Theorem 4. *For any store to the data buffer, there is a matching load that reads its value, provided enough data is requested by the consumer.*

Formally, given a store $w^{[i]}$, there is at least one load $r^{[j]}$ such that $w^{[i]} \xrightarrow{\text{rf}} r^{[j]}$, provided the consumer side contains enough non-failed non-empty pop operations:

$$\sum_{k' \in C_{\text{NF}}} n_{k'} \geq i$$

where $C_{\text{NF}} = \{l' \mid \neg[(C, l') \text{ is failed or empty}]\}$, and $n_{l'}$ denotes the batch size argument passed to (C, l').

Theorem 5. *All (non-atomic) accesses to the data buffer are data-race-free. That is, given a store $w^{[i]}$:*

$$\forall w^{[j]},\ i \equiv j \mod m \implies w^{[i]} \xrightarrow{\text{hb}} w^{[j]} \vee w^{[j]} \xrightarrow{\text{hb}} w^{[i]}$$
$$\forall r^{[j]},\ i \equiv j \mod m \implies w^{[i]} \xrightarrow{\text{hb}} r^{[j]} \vee r^{[j]} \xrightarrow{\text{hb}} w^{[i]}$$

Please refer to Appendix A.1 for the detailed proof.

```
void give(Deque *q, void *x)
{
    var b $:=_{\mathrm{rlx}}$ q.bottom;
    var t $:=_{\mathrm{acq}}$ q.top;
    var a $:=_{\mathrm{rlx}}$ q.array;
    bool ok;
    if (b − t > a.size − 1)
        a := resize(q);
    a.buffer[b % a.size] $:=_{\mathrm{rlx}}$ x;
    q.bottom $:=_{\mathrm{rel}}$ b + 1;
}
```

Figure 4.21: C11 work-stealing deque: *give*

4.2.3 Case study: the relaxed Chase–Lev deque

This section adapts the work from [Lê et al., 2013] to the C11 memory model. Compared to the paper version, which is based on the ARMv7 inline assembly, the code, execution diagrams and proof have been updated.

Another major component of our blocking interpreter is work stealing. We relax the sequentially consistent data structure by Chase and Lev, presented in Section 4.1.2.3, by taking advantage of the various options offered by C11. The new code is shown in Figures 4.21 to 4.23.

4.2.3.1 Notions of correctness for relaxed work stealing

For any given deque, *give* and *pop* operations execute on a single thread. Concurrency can only occur between one execution of *give* or *take* in the owner thread, and one or more executions of *steal* in different threads.

Given this simple protocol is respected, the expected behavior of the work-stealing deque is intuitive: tasks pushed into the deque are then either taken in reverse order by the same thread, or stolen by another thread. We say that an implementation is correct if it satisfies four criteria. Informally, those are:

1. Tasks are taken in reverse order.
2. Only tasks pushed are taken or stolen (well-defined reads).
3. A task pushed into a deque cannot be taken or stolen more than once (uniqueness).
4. Given a finite number of push operations, all pushed values will eventually be either taken or stolen exactly once, if enough take and steal operations are attempted (existence).

We first provide some intuitive reasoning, to justify the relaxed construction.

```
**void** ∗take(Deque ∗q)
{
    **var** b $:=_{\text{rlx}}$ q.bottom − 1;
    **var** a $:=_{\text{rlx}}$ q.array;
    q.bottom $:=_{\text{rlx}}$ b;
    fence$_{\text{sc}}$();
    **var** t $:=_{\text{rlx}}$ q.top;
    **void** ∗x;
    **if** (t $\leq$ b) {
        x $:=_{\text{rlx}}$ a.buffer[b % a.size];
        **if** (t = b) {
            **var** ok, _ := cas$_{\text{sc}}$(&q.top, t, t + 1);
            **if** (**not** ok)
                x := null;
            q.bottom $:=_{\text{rlx}}$ b + 1;
        }
    } **else** {
        x := null;
        q.bottom $:=_{\text{rlx}}$ b + 1;
    }
    **return** x;
}
```

Figure 4.22: C11 work-stealing deque: *take*

```
void *steal(Deque *q)
{
    var t $:=_{\text{rlx}}$ q.top;
    var b $:=_{\text{acq}}$ q.bottom;
    void *x := null;
    if (t < b) {
        var a $:=_{\text{acq}}$ q.array;
        x $:=_{\text{rlx}}$ a.buffer[t % a.size];
        var ok, _ := $\text{cas}_{\text{sc}}$(&q.top, t, t + 1);
        if (not ok)
            x := (void *)−1;
    }
    return x;
}
```

Unlike the linearizable version, here *steal* operations that fail at compare-and-swap return a special code. This is only for simplicity in the proof, so we can refer to each iteration as a separate invocation. It does not affect our definition of progress, which ignores failed non-empty *steal*.

Figure 4.23: C11 work-stealing deque: *steal*

- Newly pushed tasks are made visible to *take* and *steal* by the increment to *bottom* in *give*. As we shall see in Appendix A.2, our C11 implementation enforces this by placing a release fence before the update of *bottom*, guaranteeing that the pushed element can not be stolen before *bottom* is updated.

- Taken tasks are reserved first by updating *bottom*; in our code, the ; barrier placed after the update to *bottom* will ensure that it will not be concurrently stolen.

- Stolen tasks are reserved by updating *top*. The only situation where *steal* and *take* contend for the same task is when the deque has a single element left; this particular conflict is resolved through the compare-and-swap instructions in both *take* and *steal*. This scenario allowed Chase and Lev to make the compare-and-swap in *take* conditional upon the size of the deque being one, as we have seen in Section 4.1.2.3.

  The correctness of this optimization in a relaxed memory model depends on the presence of the two fences in *take* and *steal*, to ensure that at least one of the participants will have a consistent view of the size of the deque—the anti-store-buffering pattern of Section 4.2.1.2. Having just one *take* or *steal* instance seeing a consistent view of the size of the deque is enough: if it is *take*, that will force a compare-and-swap to be performed; if it is *steal*, the index reservation will ensure an empty return value.

- Finally, stolen tasks are protected from being concurrently stolen multiple times by the monotonic compare-and-swap update to *top* in *steal*. This compare-and-swap orders *steal* operations and makes them mutually exclusive. At the same time, *steal* operations that abort due to a failed compare-and-swap do not change the state of the deque.

The proof validates criteria 2 to 4 enumerated above. Since *give* and *take* never execute concurrently and *bottom* is only ever modified in one of these functions, the proof of criterion 1 does not involve reasoning about concurrency and we omit it here.

4.2.3.2 Preliminary definitions for the relaxed work-stealing deque

We note $b(n)$ (for $n \in \mathbf{N}$) the sequence of values taken by the variable *bottom* over the course of the program, according to the modification order relation. Initially $b(0) = 0$. Since all *give* and *take* operations occur in a single thread, and *steal* operations never alter the value of *bottom*, the elements of $b(n)$ correspond to writes to *bottom* in program order within the *give* and *take* operations. Similarly, we define $t(m)$ (for $m \in \mathbf{N}$) the sequence of values taken by the variable *top*. We assume $t(0) = 0$. Furthermore, since all writes to *top* are from compare-and-swap instructions, which are sequentially consistent with each other, and all such compare-and-swap instructions increment *top* by one, $t(m)$ is monotonically increasing, and such that $t(m) = m$.

For each index i, we define the sequence $x_i(v)$ (for $v \in \mathbf{N}$) of successive values given to the (abstract) element at index i in the deque by the last (non-resize) write $w^{a[i]}$ of a *give* operation, regardless of the address a of the underlying array. Only the last such write is called **significant** as it induces a new value to x_i, while writes due to resizing do not. For all $i \in \mathbf{N}$, $x_i(0)$, the value before the first significant write to index i, is undefined: $x_i(0) = \bot$. Similarly, a read is significant if it occurs in a successful instance of *take* or *steal*.

Lemma 23. *For all $i \in \mathbf{N}$, the writes producing the $x_i(v)$ are totally ordered by happens-before.*

Proof. Trivial since there is only one producer thread calling *give*. □

We define the **reads-from-far** relation (noted *rff* in superscript) as follows. A read operation r reads from a far write w if the two are related by a chain of copies (alternating reads-from and sequenced-before edges). This is just syntax to hide the effects of resizing.

4.2.3.3 Action structures in the work-stealing deque

We consider the three operations of the work-stealing algorithm: *take* (Figure 4.25), *give* (Figure 4.24) and *steal* (Figure 4.26). Each of them exhibits different execution paths depending on control flow. In formulas, the *bottom*, *top*, and *array* variables are denoted by their initials in superscript, and the variables b, t and a, respectively, when we mean the corresponding memory values. n_a denotes the array size of a. When indexing a, the size modulo is implicit.

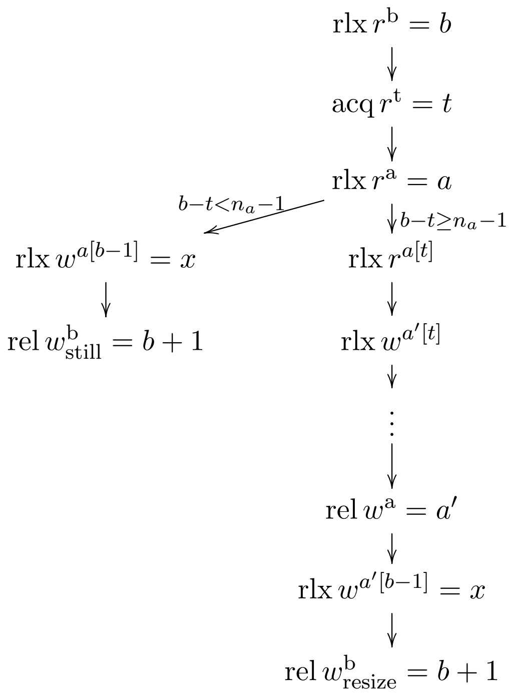

$\mathrm{rlx}\, r^{\mathrm{b}} = b$
$\mathrm{acq}\, r^{\mathrm{t}} = t$
$\mathrm{rlx}\, r^{\mathrm{a}} = a$
$b-t<n_a-1$
$b-t\geq n_a-1$
$\mathrm{rlx}\, w^{a[b-1]} = x$
$\mathrm{rel}\, w^{\mathrm{b}}_{\mathrm{still}} = b+1$
$\mathrm{rlx}\, r^{a[t]}$
$\mathrm{rlx}\, w^{a'[t]}$
$\vdots$
$\mathrm{rel}\, w^{\mathrm{a}} = a'$
$\mathrm{rlx}\, w^{a'[b-1]} = x$
$\mathrm{rel}\, w^{\mathrm{b}}_{\mathrm{resize}} = b+1$

Figure 4.24: Action structures of the *give* function

For *take* and *steal*, we say that an instance of the operation is successful if it returns one element; otherwise (including if it returns empty) it is considered failed.

4.2.3.4 Proof of the work-stealing deque

We build on a precise analysis of all the possible executions of arbitrary invocations of the algorithm. Individual lemmas strive to narrow down the set of possible candidates, based on properties of the algorithm and the memory model. To that end, we pinpoint specific subgraphs of an execution that cannot occur together in the same consistent execution. We then show that all incorrect executions, such as those containing two instances of *steal* reading the same value added by a single instance of *give*, cannot have consistent executions and, as such, cannot happen.

The proof is divided into five parts. In Section 4.2.3.3 we describe all the possible action structures for each of the three operations (*give*, *take* and *steal*), following the control flow of the code. In Appendix A.2.1 we show that the succession of dynamic arrays built by resizing can be abstracted as a single sequence of unique abstract significant values, independent of resize operations. Lemma 34 establishes criterion 2 (well-defined reads). In Appendix A.2.2 we build on the previous abstraction to prove Theorem 6, pertaining to the uniqueness of elements taken and stolen, which corresponds to criterion 3 (uniqueness). Finally, in Appendix A.2.3, we rely on all previous results to prove Theorem 7 establishing criterion 4 (existence): the existence of matching *take* or *steal*

$$\begin{tikzcd}
 & \mathrm{rlx}\, r^{\mathrm{b}} = b \arrow[d] & \\
 & \mathrm{rlx}\, r^{\mathrm{a}} = a \arrow[d] & \\
 & \mathrm{rlx}\, w^{\mathrm{b}}_{\mathrm{first}} = b-1 \arrow[d] & \\
 & \mathrm{sc}\, f \arrow[d] & \\
 & \mathrm{rlx}\, r^{\mathrm{t}}_{\mathrm{first}} = t \arrow[dl, "b-t\leq 0"'] \arrow[d, "b-t=1"'] \arrow[dr, "b-1>1"] & \\
\mathrm{rlx}\, w^{\mathrm{b}}_{\mathrm{empty}} = b & \mathrm{rlx}\, r^{a[b-1]} = x \arrow[d, "t\neq t'"'] \arrow[dr, "t=t'"] & \mathrm{rlx}\, r^{a[b-1]} = x \\
 & \mathrm{rlx}\, r^{\mathrm{t}}_{\mathrm{loss}} = t' \arrow[d] & \mathrm{sc}\, s^{\mathrm{t}} = t \rightarrow t+1 \arrow[d] \\
 & \mathrm{rlx}\, w^{\mathrm{b}}_{\mathrm{loss}} = t+1 & \mathrm{rlx}\, w^{\mathrm{b}}_{\mathrm{win}} = b
\end{tikzcd}$$

Figure 4.25: Action structures of the *take* function

$$\begin{tikzcd}
\mathrm{rlx}\, r^{\mathrm{t}}_{\mathrm{first}} = t \arrow[d] & \\
\mathrm{sc}\, f \arrow[d] & \\
\mathrm{acq}\, r^{\mathrm{b}} = b \arrow[d, "b-t>0"'] & \\
\mathrm{acq}\, r^{\mathrm{a}} = a \arrow[d] & \\
\mathrm{rlx}\, r^{a[t]} = x \arrow[d, "t=t'"] \arrow[dr, "t\neq t'"] & \\
\mathrm{sc}\, s^{\mathrm{t}} = t \rightarrow t+1 & \mathrm{rlx}\, r^{\mathrm{t}}_{\mathrm{loss}} = t'
\end{tikzcd}$$

Figure 4.26: Action structures of the *steal* function

operations for every pushed element, under the appropriate hypotheses.

Theorem 6. *Given a worker thread executing a sequence of* give *and* take *operations, and finite number number of thief threads each executing* steal *operations, all against a same deque, if X and Y are two distinct successful instances of* steal *or* take*, then:*

$$\forall r = x_i(v) \in X,\ \forall r' = x_{i'}(v') \in Y,\ i \neq i' \vee v \neq v'$$

Theorem 7. *Consider a worker thread executing a sequence of* give *and* take *operations, and a finite number of thief threads each executing* steal *operations, all against a same deque. If the number of* give *is finite, then all threads reach a stationary state where $b = t$ in a finite number of transitions, and for all $x_i(v), v > 0$, there exists a unique significant read $r = x_i(v)$ in some thread before the stationary point.*

Please refer to Appendix A.2 for the detailed proof.

4.2.4 Further relaxation

4.2.4.1 Relaxed passive waiting

The passive waiting mechanism explained in Section 4.1.3 also offers opportunities for relaxed optimization. As stated in Corollary 8, waiting on processes depends on the interplay between complete transitions, *putToSleep* and *awaken*. Each side performs a two-phase routine: on one side, a process is put to sleep, then the channel is double checked for progress; on the other side, the channel is updated, then it is checked again, this time for any dormant neighbor.

This reveals a classic anti-store-buffering pattern.[6] Correctness relies on the fact that double checking the channel is going to yield the change made on the other side, if any. In C11, it translates to the presence of a sequentially consistent fence between the two parts—the modification and the test—on each side.

The use of fences instead of qualified statements allows for some flexibility in placement. In particular, it is possible to batch multiple modifications together before the fence, and the corresponding tests after, instead of expending one fence for each pair. As it happens, this meshes very well with the idea of lazy waiting, which already does exactly that on the awakening side.

The reason this is not as easily implemented on the sleeping side has to do with ownership. We recall that this part of the construction is unsafe under delays, meaning we are allowed to assume exclusivity on some objects. For performance, in the primary process scheduler, we rely on exclusively owning a task in order to be able to move it between sets. Therefore, it is difficult to postpone putting a process to sleep. We would either need to hold onto its ownership for longer than desirable, or have to release it

[6] More so than the one in the Chase–Lev deque of Figures 4.22 and 4.23, where the modification of *top* is not in the invocation of *steal* where the fence occurs.

back to an inappropriate set and thus risk it being taken up by another unaware thread, wasting computing resources.

The downside of this optimization is quite obviously that it breaks some level of modularity. The fences need to be lifted up to the level of the scheduler itself, instead of staying isolated in a few specialized functions that deal with passive waiting specifically.

4.2.4.2 Relaxed non-blocking core interpreter

Although it may not be immediately apparent to the reader, the algorithms presented in the previous chapter were in fact designed with relaxation in mind, from the start. This is partly due to historical motivations—the blocking relaxed interpreter came first—and partly due to the authors' prior background in high-performance low-level programming.

The specialized non-blocking channel data structure, in particular, has the potential for good relaxed performance. Micro-updates to monotonic buffers only require release–acquire semantics, if we are willing to accept some intermittent delays in propagation of information and values, in the same way as in the single-producer single-consumer queue of Section 4.2.2. There is no denying that block swapping at the macro level will be costly; however, it can be mitigated by having longer buffers than strictly necessary. This will often results in having a few standard buffer sizes that suit all channels, which has other advantages, such as easing memory management (e.g., less fragmentation, and the ability to use reference counting if we so desire) and preventing false sharing with proper padding. Again, we reiterate that unlike purely batch-oriented algorithms, ours do not require Kahn processes to deal in large arrays of values at a time. The size of monotonic buffers only influences how often we need to replace them, not how much data is needed before communication can be established: at the fine-grained granularity, synchronization is ensured by release–acquire.

These points are only meaningful because they are complemented by process caching, which allows a worker thread to operate on a private copy of a process state as long as only micro-operations are involved—we only require the shared state to be updated when a macro descriptor changes. Once cached, block pointers need not go through the expensive multi-safe-read protocol, as the thread locally stores a consistent view of both the process and the associated monotonic channel chunks. Although the code is peppered with tests that recheck the value of the various cached pointers, those are entirely optional. Indeed, each test returns whether we need to reload a specific changed shared variable. However, they only serve as short-circuit device, to cut redundant paths before we reach the sole authoritative compare-and-swap operation on the shared process state. We can therefore relax them all, as their results do not matter, for correctness purposes, as long as we ensure that they do not wrongly tell us to reload a same value, as this would impede progress. This is not a problem with release–acquire synchronization; some care must be taken if we want to use fully relaxed C11 instructions, to establish happens-before relations that forbid reading pointers older than our cached versions.

4.2.5 Integration and perspectives

Sections 4.2.2 and 4.2.3 above provide ad hoc correctness criteria for two basic components of our system, through Theorems 3, 4, 6 and 7. However, most of our existing work has been in a linearizable environment, building linearizable objects. Thus, a most legitimate question to ask is how we intend to blend them together into a single working implementation—actually two of them depending on whether we go for the blocking or the non-blocking route.

4.2.5.1 Abstraction in the C11 memory model

In a sequentially consistent universe, linearizability is both an abstraction principle (i.e., the rules under which one routine can be substituted for its specification) and a well-understood specification method (sequential procedures as specification). A good place to start with our reflection is the work of Batty et al. [2013], which establishes an abstraction theorem for the C11 memory model. Below, we briefly provide an overview of the technique they propose, then we explore its potential application to our Kahn process library.

We recall that, in sequential consistency, one interpretation of classical linearization tells us that for each authorized execution of the linearizable object (the **library implementation**), there exists at least one equivalent sequential run of the specification (the **library abstraction**). The notion of equivalence is then defined by two criteria:

- any return-to-call ordering in the original history must be enforced in the linearization;

- and the histories must have the same interface—call and return—actions.

Intuitively, the first rule guarantees that any ordering enforced by the client—which can only be done outside invocations hence return-to-call—is honored by the abstraction. The second property ensures that any potential client observes the same values out of the library when putting in the same data. In a sense, they describe two sides of a same coin: the library controls what happens inside method bodies, while the client oversees everything that may occur in between.

In C11, things are similar, in spirit. Instead of concurrent interleaved histories, we have execution graphs made of C11 primitive actions (e.g., load, store, compare-and-swap, fences) and relations (e.g., happens-before, reads-from, sequentially consistent, modification). We still quantify over all possible sets of obligations in the form of return-to-call edges (first criterion), only this time, the set is only made of those constraints that are actively enforced by client code. While this may sound trivial, it means we cannot assume that $a \xrightarrow{\text{hb}} b \vee b \xrightarrow{\text{hb}} a$.

The sequential specification run becomes a standard C11 execution, since unlike sequential consistency, there is no easy (or uniquely agreed upon) way to define what it means to replace a concurrent invocation by an atomic instance of anything. For abstraction to apply, we must evidently still have identical values at the interface (second criterion). Is that all there is to it?

Unsurprisingly, the answer is no. In classical linearizability, the sequential history provides not only values but also order between invocations. From the point of view of the caller, this translates to call-to-return guarantees (since whatever happens in the middle of a method is not directly observed by the client). The set of possible linearizations is indicative of what interleaving the client may expect. For example, in a stack or in a queue, the push operation always precedes the matching pop, so a user of the library can count on that fact to derive useful patterns, such as message passing.

In C11, the situation is slightly different. Whereas the ordering of method instances is implicit in the sequence of a classical linearization, in C11, the library abstraction is now a normal execution graph. The set of invocations does not form a sequential history, and thus only certain method calls are **guaranteed** to happen before certain return actions. This is formally captured as a set of call-to-return happens-before edges. Again, note that in a relaxed model, the absence of such a relation does not imply the opposite: a call that does not happen before a return does not necessarily happen after either.

Keeping with this idea, in addition to guarantees, the library may also simply **deny** the client the possibility of forming certain return-before-call edges, which is essentially a weaker version of the above guarantees. It means that such executions of the specification are not compatible with clients that would enforce such relations. In practice, denial is mainly the work of sequentially consistent relations, which, as we have seen in Section 4.2.1.2, do not imply happens-before but are nevertheless compatible with it. There are therefore three possible relations between two invocations: one call happens before the other return, one call cannot happen after the other return (implied by the previous case), or library-neutral. Each library-side constraint restrains the number of possible executions of the client. Together with the set of interface actions, these are considered by Batty et al. [2013] to define a history in C11.

They offer two abstraction theorems, depending on the subset of C11 in use, whether it includes the infamous relaxed atomic instructions of C11 or not. For the simpler model, without relaxed, the concrete library can be replaced by the abstract version if the implementation guarantees and denies at least as much as the specification. This is intuitive, as by adding more edges from the library, we may only reduce the number of possibilities from the abstraction, never allow something that is impossible in the specification. Due to self-satisfaction cycles—which we briefly mentioned in Section 4.2.1.3—the same does not hold true when relaxed accesses are enabled. In that case, the guarantee sets on both sides must be equal.

In addition to this, something that is often taken for granted in informal sequentially consistent arguments about composition, but becomes more prevalent in the C11 world, is non-interference between the components, we touched upon in the beginning, Section 2.1.1. The reason why this specific issue is raised in the context of C11 is because, in the presence of relaxed operations, memory isolation between the client and library needs to be enforced at the level of the implementation instead of the abstraction.

It is beyond the scope of this document to explain precisely why such differences exist. However, practically speaking, by now, we should be well-aware of the simple reality that our own algorithms in Sections 4.2.2 and 4.2.3 make heavy use of those relaxed instructions—although always in conjunction with fences or release–acquire guards.

Does this mean we need to use the weaker version of the theorem? Yes and no. In truth, it depends on what clients we allow. The main issue with relaxed atomic instructions is that, counter-intuitively perhaps, the self-satisfaction pattern gives too much observational power to clients to distinguish between unwary libraries.

Even so, it should be noted that an abstraction theorem in itself—even in its weak form—is a very powerful device, as it effectively allows us to reason on a different program, by replacing the implementation with its abstraction (with the caveat that, in the case of relaxed abstraction, non-interference needs to be proven on the original program). If we can isolate low-level uses of relaxed instructions and replace them with specifications that do not contain any, then we get to use the stronger version of the theorem on any higher-level abstractions. Putting these ideas together, a possible proof strategy could be the following:

1. Do a whole-program proof of non-interference on the implementation, so we can apply the weak theorem.

2. Apply the weak theorem to every component that uses relaxed operations separately to prove it equivalent to a specification that does not involve any.

3. Substitute those into the original program.

4. Apply the strong theorem to prove any remaining abstractions.

Admittedly, this may not be possible. For starters, it requires that every component cooperates: external libraries outside of our control must provide a non-relaxed specification, or we need to write one ourselves.

Also, not every occurrence of a relaxed operation may be abstracted. However, in many cases, actual relaxed instructions are constrained by algorithmic hypotheses or contracts on the interface that must be satisfied by clients.

For example, in the single-producer single-consumer queue of Section 4.2.2, the only relaxed accesses occur on the same-side index variable, which is guaranteed exclusive to the current thread, due to the interface contract. There is therefore a single write that can be read from, each time. Changing those to acquire does not introduce new guarantees, since any would-be synchronization is already covered by same-side happens-before.

This technique, however, unfortunately meets with some difficulties when dealing with sequentially consistent instructions. In theory, we would want to go from a release–acquire implementation to a sequentially consistent abstraction—so that we can apply classical linearization over the resulting sequentially consistent program.

To continue with the queue example, the express purpose of release–acquire over sequentially consistent index accesses is to allow the machine to decide when opposite indices are made available for reading. In practice, we want to allow both consumer and

```
N front;
N back;
int data[$\infty$];

bool push(int x)
{
    if (nondet())
        return false;
    data[back++] $:=_{\text{sc}}$ x;
    return true;
}

bool push(int *x)
{
    if (nondet() or front $\geq$ back)
        return false;
    *x $:=_{\text{sc}}$ data[front++];
    return true;
}
```

Figure 4.27: Non-deterministic sequentially consistent single-producer single-consumer queue specification

$$\begin{tikzcd}
\text{push} \arrow[r, "\text{sb}"] \arrow[d, "\text{hb}"'] & \text{push} \arrow[d, "\text{hb}"] \arrow[dl, dashed, leftrightarrow, "\text{sc}"] \\
\text{pop} \arrow[r, "\text{sb}"'] & \text{pop}
\end{tikzcd}$$

Figure 4.28: Simple non-deterministic queue example

producer to bump their indices without immediately informing the other side; in essence, it is a form of on-write caching, symmetrical to the on-read caching we do manually. Hiding this behavior behind non-determinism, we get the specification in Figure 4.27.

However, we cannot apply the C11 abstraction theorem. Replacing the release–acquire pair with sequentially consistent primitives introduces edges in the deny set, due to the sequentially consistent total order. For example, consider Figure 4.28, where we have represented four successful invocations of queue methods. Each pop reads from the above push. If this is an execution of the release–acquire implementation, then the second push does not synchronize with the first pop one way or the other.

There is no execution of the sequentially consistent code of Figure 4.27, however, which does not also add a sequentially consistent edge between the first pop and second push, denoted as a dashed arrow. That edge denies the opposite return-to-call relation from the client. Without the sequentially consistent C11 instruction set, we can only add to the deny set through happens-before; yet, we certainly do not want our second push to have to wait for the first pop to take effect, or the other way around. Therefore, the release-acquire queue implementation is not abstracted by our non-deterministic specification.

Perhaps it was naive of us to expect as much, in the first place. It could certainly be argued that it was never the purpose of the C11 standard to offer such functionality. However, we find the difficulty of mixing classical linearizability results with relaxed components to be quite restrictive and problematic at scale, where most higher-level components have no need for the minutiae of relaxation.

4.2.5.2 Systematic uses of relaxed instructions

The previous example of a relaxed read statement being converted to acquire is actually not an isolated instance. Throughout our journey in the land of C11 concurrent algorithms, we have come to distinguish a few systematic use cases of relaxed instructions that we now explain.

Exclusive read The most trivial usage of relaxed is for exclusive access on read. This is the case for same-side index reads in both the queue (*front* for the consumer, *back* for the producer) and the work-stealing deque (*bottom*, *array* pointer and elements for each owner). In this scenario, we know that every modification of the target location either happens before or after the current load operation. This kind of relaxed access can be replaced by an explicit local cached copy of the variable, using non-concurrent accesses, with updates changing both.

Speculative part of a double check Lock-free programming is rife with this sort of pattern, where a value is preventively loaded for an operation, only to be confirmed later by a second "proper read." As we have explained in the previous chapter, the correctness of this kind of double checks lies in the fact that the speculative read basically serves as an oracle: any value could be returned, and the algorithm would still proceed safely (although it may not make progress). One obvious use is to read the expected value of an upcoming compare-and-swap, as is done in *take* and *steal* for work stealing.

Redundant acquire fence This category encompasses those cases where an acquire qualifier can be removed due to being redundant with another operation. Case in point: the first relaxed read in the *steal* deque method. It is followed by a sequentially consistent fence, and an acquire fence on the load would be flagged as locally redundant.

The other relaxed read in *steal* is more subtle. It occurs on the array element, and is followed by compare-and-swap. There are two cases. If the function succeeds, then any information that would be acquired by the data read is also carried by the previously loaded *bottom* variable, which is updated by the producing thread after the element is written. If the function fails, then the item value is discarded and everything proceeds as if nothing was read in the first place: a would-be specification of the deque may not include such an operation in its failing branch at all.

Once those cases are taken out of the picture, the only remaining uses of relaxed are writing to array elements in *give* and updates to the *bottom* variable in *take*. Those are more subtle and depend on the algorithm.

For array item updates, this is a property of *steal* that makes it so it never returns an outdated element, therefore, the following release from writing to *bottom* carries onto any successful *steal* invocation, which relieves the need for another release operation on the item itself.

For owner index updates in *take*, the problem is different. Since the indices merely guard the content handed over by the user to *give*, we do not need to establish happens-before between *take* and *steal*. Therefore, we are content with letting the updates fall into the release sequence of the previous instance of *give*, instead of every one of them being a release store. This is a concept we have not explained in Section 4.2.1. Quite simply, it is similar to having a release fence before the first write in a same-thread sequence: everything that comes before the fence is released along with any of the subsequent writes when it is acquired.

For those two cases, especially the last one, writing an abstract specification in C11 without using relaxed would no doubt require non-deterministic failure to simulate the cases where no happens-before edge occurs between *take* and *steal* even though they do communicate and share indices.

4.2.5.3 Correctness in the C11 memory model

From the realization that current abstraction theorems in C11 are unlikely to help us integrate relaxed content directly into our existing linearizable code, we are left with several suboptimal solutions.

Of course, the basic, straightforward and rather disappointing solution is to simply use the sequentially consistent subset of C11 for everything. It is good that it exists so at least we know there exists a slow but correct version of our algorithms.

At the opposite end of the spectrum, we have the frontal approach. We simply go ahead and formalize any correctness properties we want on the C11 code directly. Needless to say, this has scaling issues, both in terms of writing the correct specification for multiple layers of components, and human horse power needed to produce the proofs—informal as they may be—with enough confidence. The former is difficult without the guidance of an abstraction principle or reasoning framework. The latter is problematic due to the size of the model and the code, and again the lack of proper tools to deal with them.

A third option, which is currently unavailable, would be to find a suitable subset of the C11 memory model that allows linearizable reasoning without being as limiting as the so-called sequentially consistent native instruction set.

Finally, the approach we consider the most promising is the recent development of program logics adapted to relaxed memory models, such as [Vafeiadis and Narayan, 2013, Turon et al., 2014, Lahav and Vafeiadis, 2015]. We believe this direction provides promising techniques to assist in writing proper specifications and proofs, much as we would do in a frontal engagement of the problem, albeit hopefully with less need for brute force.

4.3 Applications

4.3.1 Data-flow and task-based parallelism

We conclude our study of Kahn process networks with a look at what kind of applications may benefit from our work. We distinguish between what we consider typical data-flow use cases, and other more general parallel programming scenarios, many of which have been briefly introduced in Section 4.1.1.1, when discussing the various scheduling constraints under which one can operate.

Data-flow applications include those for which a description as Kahn process networks already exists or can be naturally formulated. As we have already mentioned, stream-oriented languages such as StreamIt [Thies et al., 2002], and synchronous data-flow languages such as Lustre [Caspi et al., 1987] provide a good reservoir of potential candidates. Automated compilation from such high-level programming languages to state machines suitable for execution on our run-time library would certainly be a worthwhile project.

More general concurrent programming models with streams include Y-API [de Kock et al., 2000] and the Go language. In addition to single-producer single-consumer channels, Go supports non-deterministic multi-producer multi-consumer queues. Both also allow non-determinism through the use of waits on multiple objects (similar to the *select* Unix call) within processes.

Non-data-flow applications are those that exhibit parallelism expressed using a different metaphor: e.g., fork–join or task-based parallelism.

- In a fork–join design, as found in the Cilk language [Blumofe et al., 1995] and its descendants, sequential execution is allowed to split (fork) and then wait for all the parts to finish (join) before resuming. This was the original model of work stealing as introduced by Blumofe and Leiserson [1999].
- Although there is no single accepted definition of task-based parallelism, by far the most common idiom—implemented in [Planas et al., 2009, Pop and Cohen, 2013] as well as OpenMP—takes the following shape. Programs are described as direct acyclic graphs: the vertices represent sequential work units, while the edges are dependency constraints. The graph itself is usually built sequentially by a control thread (or task), which, depending on the exact implementation, has various ways of synchronizing with (e.g., waiting for) the tasks it spawns. Some systems permit recursive graphs, in which single tasks may themselves split into a private subgraph. This provides a kind of hybrid with fork–join parallelism, in which parent nodes are offered more control over how their children execute (through dependencies), instead of simply splitting and waiting.

While it could be argued that task graphs are similar to Kahn process networks, they differ in some important ways. Most visibly, Kahn processes are stateful and communicate through channels, whereas tasks in a task-based environment are one-shot activities that have a fixed set of one-time dependencies. In most implementations, it is possible to pass (implicit) memory ownership through such dependencies; some, such as OpenStream, also feature explicit data-carrying objects.

It is true, however, that the two paradigms are convertible, in a sense. On the one hand, finite direct acyclic graphs of tasks can be systematically translated to Kahn process networks by mapping tasks to processes one for one, and dependencies to single-item channels. Each process simply reads from all its input channels, executes the task it has been assigned then pushes the result on all connected output channels. Conversely, Kahn processes can be unrolled, as shown in Figure 4.29, so that each transition defines a task and individual channel indices correspond to a dependency edge between producers and consumers.

The translation problem is complicated by dynamic creation of processes and tasks: recursive process networks, as opposed to the sequential control model. This is a concern mostly as regards memory management. When do nodes and edges cease ot be eligible for further graph extensions—and can thus be recycled? When can we start executing?

In classical Kahn process networks, this problem is addressed by the recursion mechanism. Only new processes can have their input and output channels bound at creation time, and channels are scoped by the reconfiguration block they belong to.

In sequential-control task graphs, the control task similarly reigns over creation and decides what lives, what dies, and when. The main difference lies in the fact that child

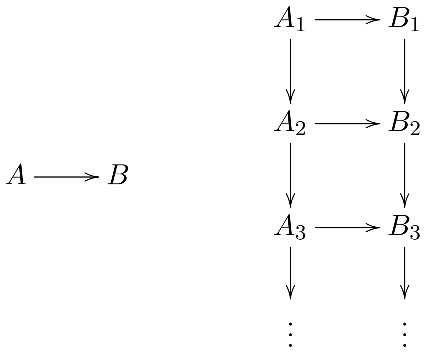


Figure 4.29: Kahn process to individual tasks

nodes can be added and bound incrementally, whereas the Kahn *reconfigure* statement is atomic. This design discrepancy is closely related to the necessity, in the task-based model, of active recycling: any infinitely running program will generate an infinite number of tasks and dependencies. Compare this to Kahn processes, which are mostly expected to stabilize at a given graph structure and continue looping through the same channels, reusing buffers and states. Therefore, not only does the control thread incrementally creates, it must also incrementally destroy.

The same effect can be achieved by extending the Kahn process language to allow incremental reconfiguration, whereby channels can stay unbound on one side while already configured processes start executing. We must also add to the run-time library the ability to delete process and channel objects, once they terminate or both of their ends close, respectively. This is not exactly trivial in the non-blocking algorithm, which uses a tree bin to manage the memory associated with the network itself (as opposed to individual state and buffer blocks). However, our blocking implementation does offer such fine-grained control, coupled with a more precise memory management system.

With these similarities and differences hopefully clarified, we can now focus on a couple of concrete examples and results.

4.3.2 Experimental results and the libkpn blocking implementation

Our experimental evaluation is based on our blocking run-time library—**libkpn**, which acts as a kind of performance benchmark for our future Kahn process implementation endeavors.

Indeed, the blocking library is geared toward high performance. It employs a single primary scheduler based on the relaxed work-stealing deques described in Section 4.2.3 and the lazy passive waiting algorithms of Section 4.1.3, relaxed according to Section 4.2.4.1.

Worker threads acquire exclusive access to Kahn processes by removing them from shared sets (e.g., the work-stealing deque) and keeping them from being seen by other threads. Workers run the process state machines and carry out any communications as needed, through the cached single-producer single-consumer channel of Section 4.2.2.

As explained above, the blocking run-time library supports incremental reconfiguration, as well as partial garbage collection of channel and process objects as they become unused. This allows for easy emulation of task-based run-time libraries. In fact, libkpn offers an emulation layer, KOMP, for the OpenMP task subsystem, where OpenMP directives can be mapped one-for-one to corresponding function calls.

Compared with a would-be complete non-blocking implementation, this version requires less bookkeeping (e.g., safe reads), has less overhead in terms of costly instructions (no need for compare-and-swap unless sleeping or contending for the last item in a deque). In particular, both state updates and channel operations can be done entirely in-place, without the need to constantly swap in new blocks. In a way, it represents an upper bound on the performance that can be expected of the lock-free implementation in the unlikely event that:

- monotonic buffers are infinite (or large enough to hold the entirety of a finite execution), such that there is no need to ever replace descriptors;

- and processes are distributed among worker threads with no redundant computation, at all times.

For more information on the concrete C11 source code of the libkpn library, please refer to Appendix B.

Direct comparison with existing implementations of such data-flow models is possible if we limit ourselves to the subsets commonly supported. Preliminary measurements on simple benchmarks point toward performances roughly on par with StreamIt for the static subset and Go for Kahn process networks on small configurations (few processors). On larger systems, our implementation scales better.

We now concentrate on a couple of in-depth examples drawn from the task-parallelism literature.

4.3.3 Case study: matrix factorizations

Matrix algorithms make an interesting case study for several reasons. They are well-defined problems with well-known solutions, including parallel implementations readily available in the form of libraries such as LAPACK (and compatible replacements). Furthermore, they have been shown to utilize task-based concurrency well in a shared-memory setting. [Buttari et al., 2009]

The basic idea behind parallel matrix programming is tiling. Most common matrix transforms have a block version, where operations on individual elements are replaced with analogous routines on submatrices. This offers control over the granularity at which sequential processing unfolds. Therefore, we can represent the calculation as a number of interdependent steps: each step reads from one or more input tiles, computes some values, and writes the results in one output tile.

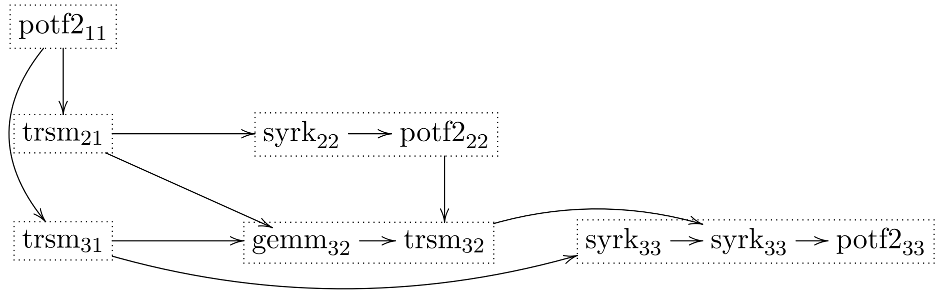


Figure 4.30: Dependency graph for a 3x3 Cholesky transform

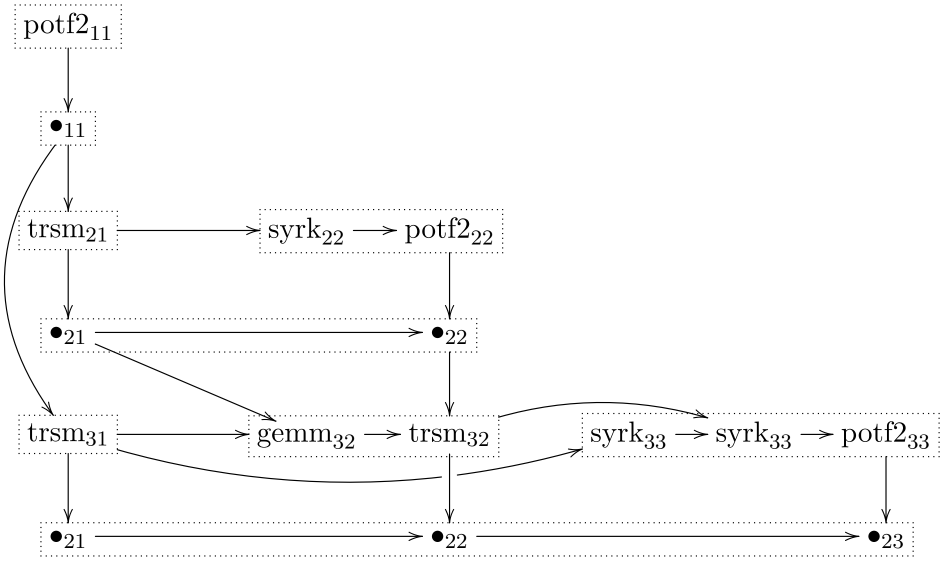


Figure 4.31: Row aggregation for the Cholesky transform

4.3.3.1 The Cholesky transform

Our first example is the Cholesky transform. Figure 4.30 shows its action on a 3x3 block matrix; each element shown is actually a tile of unspecified size. Each step is a call names a basic matrix operation from BLAS or LAPACK. Knowledge of what each routine does precisely is not essential to the comprehension of the parallel arrangement explained here. Briefly, *potf2* is the sequential base-case Cholesky factorization; *trsm* solves a triangular matrix equation; *syrk* performs a specific case of matrix-matrix multiplication of a given matrix with its own transpose; and *gemm* is the general matrix-matrix multiplication. They are linked by arrows indicating what intermediate results are needed for each step. The ones that modify a common tile are grouped in dotted boxes; the tile indices are denoted in subscripts.

The transition to task-based parallelism is straightforward, by taking each step as a task, and arrows as dependencies; it then suffices that we choose any compatible sequential creation order. The graph has both vertices and edges in $O(n^3)$, where n is the size of the matrix (the length of one side such that it has n^2 elements in total), although popular frameworks, including StarSs, OpenMP, and Swan, all aggregate dependencies originating from the same tile, and thus have only $O(n^2)$ dependency objects.

The same algorithm can be implemented as is on our libkpn run-time system. It should be noted that, since the library is not lock-free, it allows exclusive memory ownership in Kahn processes. In the case of block matrix algorithms, this allows our implementations to work in-place on shared-memory tiles, only relying on channels to communicate dependencies, along with implicit permissions to access the tiles.

Optimizing Cholesky We now describe an enhanced Cholesky algorithm which exploits the features of Kahn process networks. Starting from the previous task-based version, we proceed to optimize task and dependency usage.

We remark that steps in Figure 4.30 only have write permissions to a single tile. Same-tile steps are therefore prime candidates for task aggregation into Kahn processes. Each process is responsible for one tile, which improves locality: those tasks will have a higher chance of running one after another uninterrupted (temporal locality), due to the bias in the primary scheduler as described in Section 4.1.2.1, and on a same worker thread (spatial locality). This aggregation is already visible as dotted boxes in Figure 4.30.

The next step is to fuse dependencies into channels, since having single-item channels in a Kahn process network is suboptimal. The basic rule in order for two one-shot dependencies to become part of a same channel is that they must be totally ordered. If $A \rightarrow B$ and $C \rightarrow D$ are two dependency edges, then they can only be merged if $A \rightarrow D \vee C \rightarrow B$. More generally, we want to find a sequence of dependencies $A_i \rightarrow B_i$ such that $\forall i \leq j,\ A_i \rightarrow B_j$. This way, it suffices to keep track of the highest satisfied A_i, in order to know which B_j can execute: we have a channel. However, due to the single-producer single-consumer nature of channels in a Kahn process network, an additional constraint applies: the A_i tasks must be part of a same Kahn process, and similarly for the B_j tasks.

In our case, with Cholesky, no natural channels appear in the task-graph representation. Tiles do not exhibit trivial channel constructs in their dependencies. Consider the process corresponding to tile $(3, 2)$. Its first task depends on trsm_{21} and its second task on $\mathrm{potf2}_{22}$. These dependencies satisfy the ordering requirement. However, they do not originate from the same tile—one is produced by $(2, 1)$ and the other by by $(2, 2)$, which precludes the use of a channel, a priori.

To realize this pattern, we need to introduce another level of aggregation on top of tile processes: at the row level. We add an artificial process representing each row of tiles, as shown in Figure 4.31. The intrinsic Cholesky dependencies are thus cut in two: tile processes first report to the row process, which then broadcasts results to further tasks as needed. Indeed, this can be seen as a recursive use of Kahn processes: first, we have only rows, then each row reconfigures into a full-fledged, fine-grained, subgraph.

While this appears to be a futile exercise at first—after all we are adding more dependency objects, the main benefit lies in that once thus aggregated, the results can be transmitted through a channel to every tiles located on lower rows of the matrix—e.g., row 2 is consumed by tiles $(i, 2)$ for all $i > 2$. Notice that, by applying this optimization, we convert what would be a product into a sum in the number of necessary objects to represent all the edges in the graph, resulting in $O(n^2)$ occupation. This result is similar to what is achieved by task-based frameworks that aggregate based on memory location—hence on tiles. However, the aggregation patterns are different.

The traditional task-based approach is to ease duplication by allowing multiple consumers for a single producer natively. This is usually done at the expense of precise garbage collection, as noted above. In models based on sequential creation in a control thread or task, it is that sequential order that determines when previous values are shadowed by newer ones. For example, in Cholesky, the result of syrk_{22} cannot be read anymore once potf2_{22} spawns. This, however, has the undesirable side effect of creating a bottleneck in the form of the creation path. The problem is well-known in the folklore: if the control thread moves too fast, then the scheduler is overwhelmed with tasks to schedule; if it moves too slowly, then workers starve. In contrast, Kahn processes and channels, by design, streamline part of the resource management, as either transitions within a single process (for tasks) or items within a channel (for dependencies).

4.3.3.2 The LU factorization

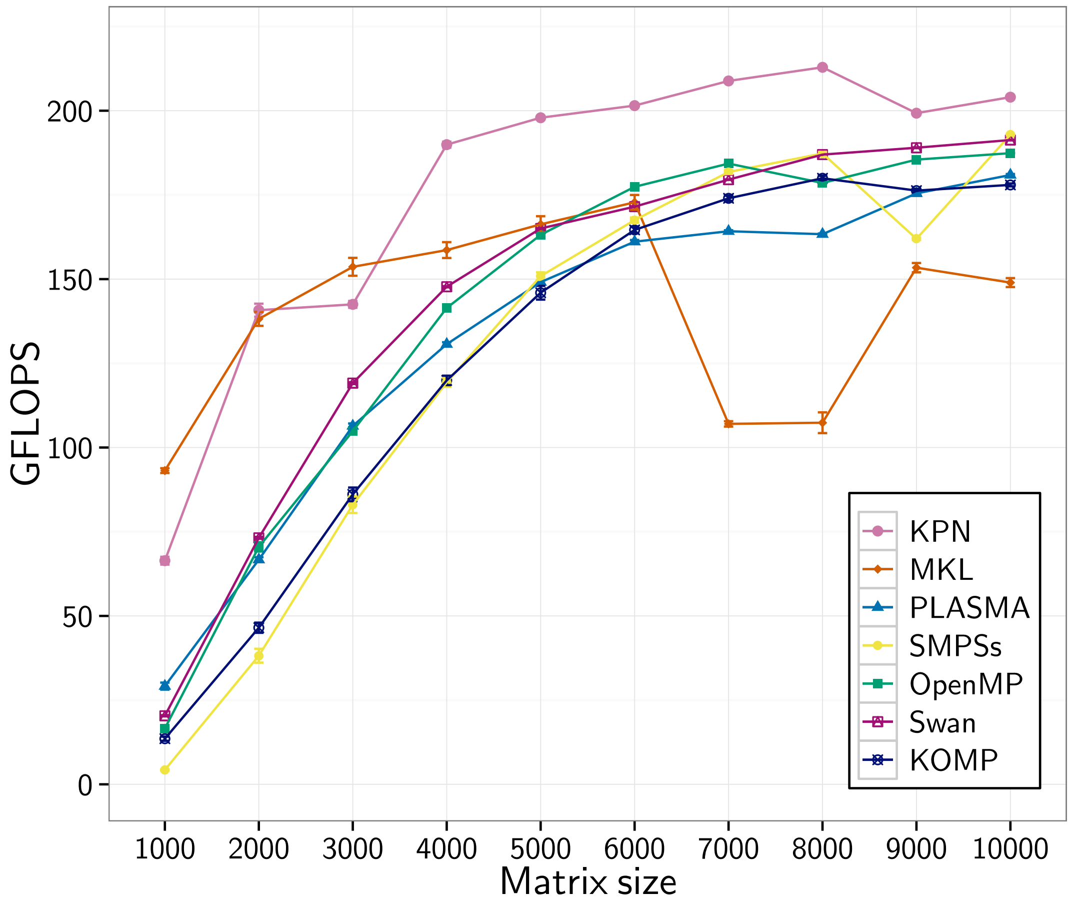
As a complementary result, we present results for a Kahn implementation of the LU factorization. Our algorithm is based on the ScaLAPACK decomposition [Choi et al., 1996], which is a tiling algorithm similar to the Cholesky parallelization shown above.

We thus apply a similar technique to the one described above, where same-tile steps are collected into Kahn processes. However, the classic LU transform also features expensive cross-tile steps: a recursive *getf2*—the LU decomposition routine in LAPACK—call applied to every column, along with permutations that span multiple tiles. Such multi-level permissions translate to transverse Kahn processes, similar to the row processes in Cholesky.

The presence of those multi-tile steps also means that there is less freedom and more imbalance in scheduling tile processes: we may need to wait for a whole column to finish, instead of pursing individual calculations off the result of a few tiles, as in the Cholesky algorithm. The combination of these factors results in overall lesser performances than for Cholesky across the board, for all implementations.

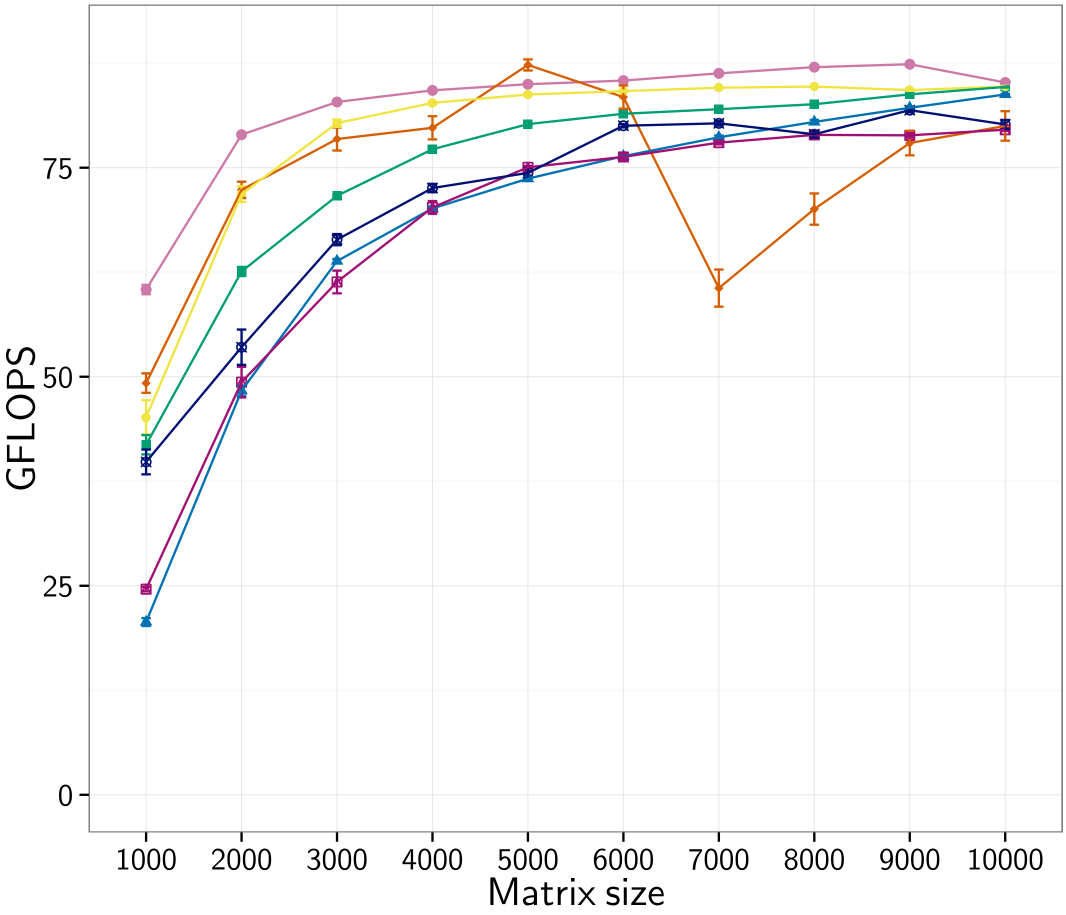
4.3.3.3 Experimental comparison

Evaluation of the Cholesky factorization We compare four task-based run-time systems running the generic tiled algorithm described at the beginning of this subsection, against our optimized Kahn version as well as two other hand-tuned reference implementations. The four generic participants are: Swan, the OpenMP 4, SMPSs (the shared-memory variant of StarSs), and our own KOMP emulation layer. The two reference implementations are PLASMA, the shared-memory parallel successor to LAPACK by Buttari et al. [2009], and Intel MKL, as shipped with their C compiler, version 14.



Twelve-core dual-socket workstation with Intel Xeon E5-2300 at 2.6 GHz; Linux 3.13.

Figure 4.32: Cholesky factorization on Xeon



Four-core desktop with Intel Core i7-2600 at 3.4 GHz; Linux 3.2.
Same implementations as in Figure 4.32.

Figure 4.33: Cholesky factorization on i7

The GNU compiler, version 4.9.1, was used for those codes that depend on it (Swan, SMPSs, OpenMP 4), while the Intel one took care of the rest. All versions were given access to the optimized sequential BLAS and LAPACK routines from Intel, in an effort to isolate the impact of parallelization. For each of those, a suitable tile size was searched beforehand, as a preprocessing step.

Figures 4.32 and 4.33 show that the four generic systems share similarly shaped performance curves, which appear to plateau for matrix sizes greater than 6000. At that point, the difference between any two implementations is a constant overhead factor on task and dependency operations, of which there are in the order of n^3, the same order as the number of floating-point matrix operations. As expected, we verify that this overhead is constant per operation and is not affected by the decision to aggregate dependencies or represent them separately.

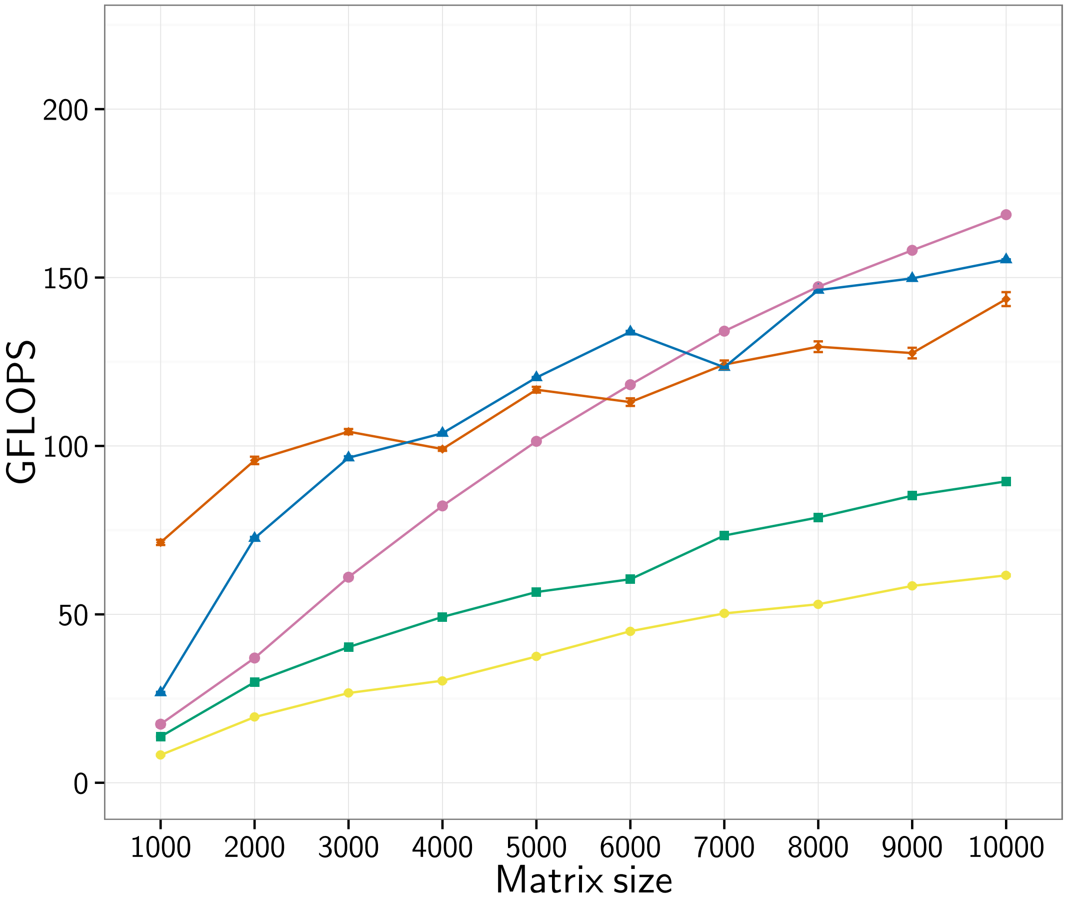
Taking OpenMP 4 as the reference point, once the curves have plateaued, the overhead of KOMP stays stable at 5% on both test machines. For simplicity, we have opted for a naive dynamic interpretation of OpenMP directives in our emulation layer, so part of the overhead can be attributed to the cost of mapping and juggling sequentially with dependency objects within the parent task. More importantly, the difference between KOMP and OpenMP is within the variations of Swan and SMPSs, which perform alternatively better and worse than OpenMP on the two platforms.

On our test platforms, our enhanced version, labeled *KPN*, consistently outperforms the competition, including the state-of-the-art reference implementations. On the bigger twelve-core Xeon machine, the difference is more visible. Our enhanced algorithm peaks earlier, exhibiting a throughput 19% higher at sizes 4000 and 5000, which progressively tapers off; still, our FLOPS count remains 5% higher than the closest alternative once all implementations plateau.

The Intel MKL results deserve their own small paragraph. Contrary to other implementations of Cholesky, the version from Intel automatically selects its own tile size. Furthermore, it is unknown what algorithms and schedulers are actually used by the library. As such, we can only speculate as to the reason of the sharp drop in performance around 7000. The most likely explanation is that smaller sizes use dedicated settings, ad hoc implementations, partially static schedules, or a combination thereof, which are abandoned at higher matrix sizes in favor of more generic but maybe less well-tuned algorithms.

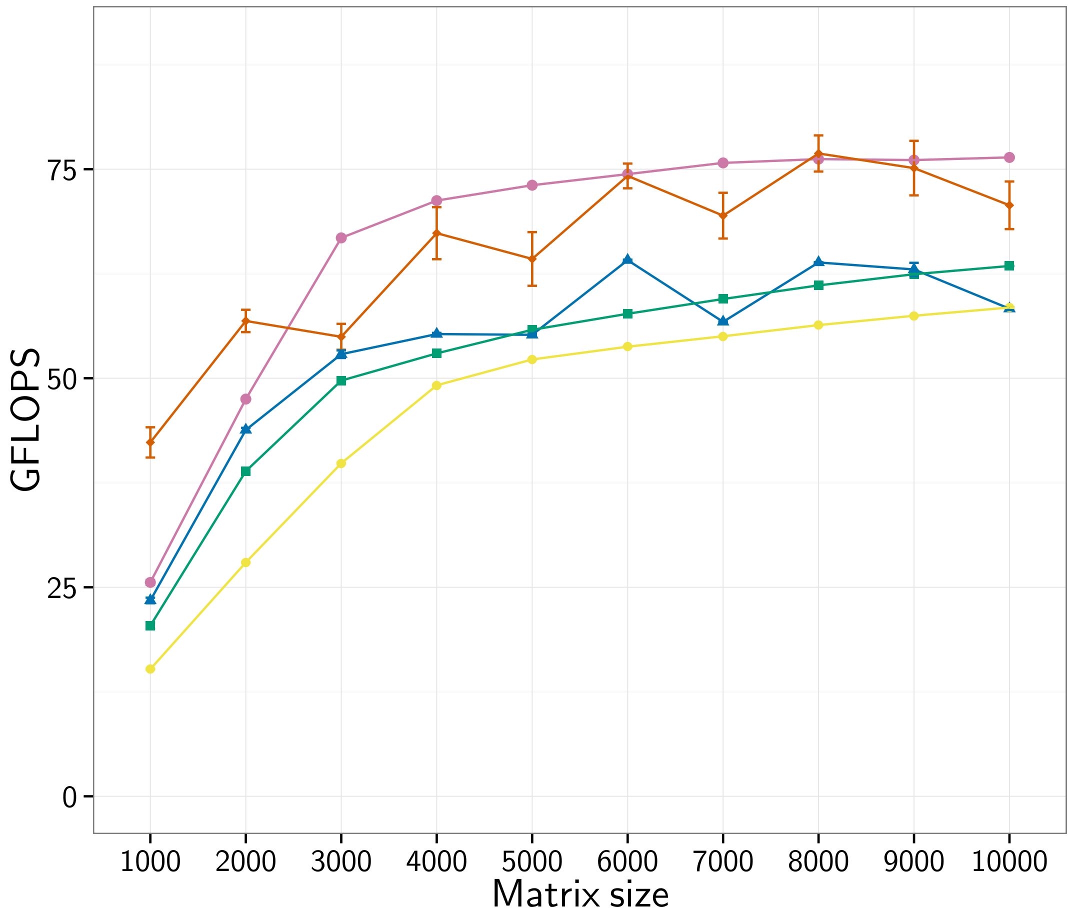
Evaluation of the LU factorization Figures 4.34 and 4.35 display FLOPS measurements for five implementations of the LU factorization: the SMPSs version and its OpenMP 4 port, the MKL and PLASMA parallel LAPACK routines, and our a Kahn process network.

Compared to Intel MKL and PLASMA, our implementation appears to be competitive on moderate and large matrices, but suffers on smaller sizes. While we applied similar optimization techniques to LU as we did to Cholesky, our implementation is missing one improvement that we believe is mostly responsible for these relatively poor results: in the LU factorization, the per-column *getf2* operation, which is fairly expensive, can execute



Same configuration as in Figure 4.32 above.
Same implementations excluding Swan and KOMP.

Figure 4.34: LU factorization on Xeon



Same configuration as in Figure 4.33 above.
Same implementations excluding Swan and KOMP.

Figure 4.35: LU factorization on i7

on large portions of the matrix sequentially. We believe it would benefit from parallelization at a finer grain, rather than being treated as a whole step.

Chapter 5

Conclusion

In this thesis, we have discussed new parallel implementations of Kahn process networks on shared-memory systems, blocking and non-blocking, in sequential consistency as well as in the C11 relaxed memory model. The lock-free algorithm has good theoretical properties as regards resource usage, contention and overhead. The blocking algorithm is simpler and achieves promising performance on practical applications, in contexts that do not require or benefit much from lock-free properties.

That being said, many topics were, however, not addressed or only partially covered. In particular, the non-blocking algorithm has not been fully implemented, and more work is still needed before we can hope to achieve a full proof of a relaxed version in the C11 memory model. This opens up a number of interesting perspectives for future work:

C11 non-blocking implementation A partial concrete implementation of the non-blocking algorithm can be readily derived from the code in Chapter 3. Aside from a few junctions here and there in the code (e.g., with regard to process and network creation), the main missing component is the lock-free memory management system described in Chapter 2. For better mainstream acceptance, more work still would be needed to expose a friendly interface to the host language. A relaxed C11 implementation is probably dependent on advances on a C11 proof (see below).

Algorithmic improvements We believe the macro queue of Section 3.7 can be simplified to remove the need for a two-word update, by inlining each half of the queue in the corresponding process state. This, however, greatly complicates both reconfiguration (which would then require a multi-word transaction) and memory management, as it conflates the permissions related to states and buffers—a single monotonic chunk could end up being referenced from multiple states. We suspect a more flexible interface to the memory management system would be required.

Dynamic worker pool A useful extension to both the blocking and non-blocking implementations is the possibility to dynamically add and remove worker threads on the fly. Such a feature would complement our lock-free properties well, seeing that we already support a fluctuating number of workers (out of a maximum) due to delays. It would mainly require adaptations from the global memory allocator (e.g., per-thread hazard pointers) and scheduling data structures (e.g., per-thread work-stealing deques).

Dynamic reconfiguration Another potential extension is the ability to reconfigure incrementally while reclaiming unused portions of the graph. This is already available in our blocking implementation, which allows for better interoperability with task-based tools, as explained in Section 4.3.1. Such a feature, however, is non-trivial in the context of our lock-free algorithm, as the process graph is currently assumed never to be freed. A move toward a more dynamic network would certainly increase the number of safe reads, and impose a new block usage strategy that allows memory to be recycled.

C11 proof Lastly, as we suggested in Section 4.2.5.3, we believe it should be feasible (although by no means easy) to prove a moderately relaxed version of our algorithms in a simplified C11 memory model featuring only release–acquire and compare-and-swap, under a relaxed program logic such as [Turon et al., 2014]. Of all the perspectives mentioned in this list, this is perhaps the most elusive yet, but also, in our opinion, the most promising. Throughout the algorithmic journey summarized in this thesis, we have had the chance to test and study several approaches (classical linearizability [Herlihy and Wing, 1987], direct use of the C11 memory model [Batty et al., 2011], and C11 library abstraction [Batty et al., 2013]) to engineering a reasonable proof—informal as it may be—of the kind of moderately large lock-free concurrent systems of which our Kahn process interpreter provides a good illustration. Although, with respect to proofs, we have not reached a definitive answer that combines all the right ingredients (e.g., lock freedom and relaxation), if anything, the experience has left us with the sense that there is much to be researched and hopefully much to be found ahead, in this direction.

Bibliography

H. G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21 (4):280–294, April 1978. ISSN 0001-0782. doi: 10.1145/359460.359470. URL `http://doi.acm.org/10.1145/359460.359470`.

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926394. URL `http://doi.acm.org/10.1145/1926385.1926394`.

M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 235–248, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429099. URL `http://doi.acm.org/10.1145/2429069.2429099`.

R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999. ISSN 0004-5411. doi: 10.1145/324133.324234. URL `http://doi.acm.org/10.1145/324133.324234`.

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. doi: 10.1145/209936.209958. URL `http://doi.acm.org/10.1145/209936.209958`.

A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 33–42, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1572-2. doi: 10.1145/2486159.2486184. URL `http://doi.acm.org/10.1145/2486159.2486184`.

A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, January 2009. ISSN 0167-8191. doi: 10.1016/j.parco.2008.10.002. URL `http://dx.doi.org/10.1016/j.parco.2008.10.002`.

P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2. doi: 10.1145/41625.41641. URL `http://doi.acm.org/10.1145/41625.41641`.

D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM. ISBN 1-58113-986-1. doi: 10.1145/1073970.1073974. URL `http://doi.acm.org/10.1145/1073970.1073974`.

J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. Design and implementation of the scalapack lu, qr, and cholesky factorization routines. *Sci. Program.*, 5(3):173–184, August 1996. ISSN 1058-9244. doi: 10.1155/1996/483083. URL `http://dx.doi.org/10.1155/1996/483083`.

E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi: Application modeling for signal processing systems. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 402–405, New York, NY, USA, 2000. ACM. ISBN 1-58113-187-9. doi: 10.1145/337292.337511. URL `http://doi.acm.org/10.1145/337292.337511`.

M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.*, 23 (2):375–382, February 2012. ISSN 1045-9219. doi: 10.1109/TPDS.2011.159. URL `http://dx.doi.org/10.1109/TPDS.2011.159`.

E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965. ISSN 0001-0782. doi: 10.1145/365559.365617. URL `http://doi.acm.org/10.1145/365559.365617`.

T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-00073-9. URL `http://dl.acm.org/citation.cfm?id=645959.676137`.

M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures. Technical report, Mountain View, CA, USA, 2002.

M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 276–290, New York, NY, USA, 1988. ACM. ISBN 0-89791-277-2. doi: 10.1145/62546.62593. URL `http://doi.acm.org/10.1145/62546.62593`.

M. P. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice* *of Parallel Programming*, PPOPP '90, pages 197–206, New York, NY, USA, 1990. ACM. ISBN 0-89791-350-7. doi: 10.1145/99163.99185. URL `http://doi.acm.org/10.1145/99163.99185`.

M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 13–26, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2. doi: 10.1145/41625.41627. URL `http://doi.acm.org/10.1145/41625.41627`.

M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1920-2. URL `http://dl.acm.org/citation.cfm?id=850929.851942`.

G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Proceedings of IFIP 74*, pages 471–475, Amsterdam, North-Holland, 1974.

G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Proceedings of IFIP 77*, pages 993–998, Amsterdam, North-Holland, 1977.

O. Lahav and V. Vafeiadis. Owicki-Gries reasoning for weak memory models. In *Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135*, ICALP 2015, pages 311–323, New York, NY, USA, 2015. Springer-Verlag New York, Inc. ISBN 978-3-662-47665-9. doi: 10.1007/978-3-662-47666-6_25. URL `http://dx.doi.org/10.1007/978-3-662-47666-6_25`.

L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, March 1977. ISSN 0098-5589. doi: 10.1109/TSE.1977.229904. URL `http://dx.doi.org/10.1109/TSE.1977.229904`.

N. M. Lê, A. Guatto, A. Cohen, and A. Pop. Correct and efficient bounded FIFO queues. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*, pages 144–151. IEEE, 2013.

N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 69–80, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442524. URL `http://doi.acm.org/10.1145/2442516.2442524`.

E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, January 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.5009446. URL `http://dx.doi.org/10.1109/TC.1987.5009446`.

P. P. C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '09, pages 78–79, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-630-4. doi: 10.1145/1882486.1882508. URL `http://doi.acm.org/10.1145/1882486.1882508`.

P. McKenney. `https://lwn.net/Articles/305782/`, 2008. Accessed on 2016-07-21.

P. E. McKenney and J. D. Slingwine. Read-copy-update: Using execution history to solve concurrency problems. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS '98, pages 509–518, October 1998.

M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 21–30, New York, NY, USA, 2002. ACM. ISBN 1-58113-485-1. doi: 10.1145/571825.571829. URL `http://doi.acm.org/10.1145/571825.571829`.

M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM. ISBN 0-89791-800-2. doi: 10.1145/248052.248106. URL `http://doi.acm.org/10.1145/248052.248106`.

P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.12.035. URL `http://dx.doi.org/10.1016/j.tcs.2006.12.035`.

P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 85–94, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835722. URL `http://doi.acm.org/10.1145/1835698.1835722`.

G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters 12(3)*, pages 115–116, 1981.

J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

A. Pop and A. Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, January 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400712. URL `http://doi.acm.org/10.1145/2400682.2400712`.

W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4. URL `http://dl.acm.org/citation.cfm?id=647478.727935`.

S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 357–368, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555261. URL `http://doi.acm.org/10.1145/2555243.2555261`.

R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 691–707, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660243. URL `http://doi.acm.org/10.1145/2660193.2660243`.

V. Vafeiadis and C. Narayan. Relaxed Separation Logic: A program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 867–884, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509532. URL `http://doi.acm.org/10.1145/2509136.2509532`.

V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 209–220, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676995. URL `http://doi.acm.org/10.1145/2676726.2676995`.

J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. doi: 10.1145/224964.224988. URL `http://doi.acm.org/10.1145/224964.224988`.

H. Vandierendonck, K. Chronaki, and D. S. Nikolopoulos. Deterministic scale-free pipeline parallelism with hyperqueues. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 32:1–32:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503233. URL `http://doi.acm.org/10.1145/2503210.2503233`.

Appendix A

Proofs of data structures in the C11 memory model

A.1 Proof of the C11 single-producer single-consumer queue

This section is reproduced in full from [Lê et al., 2013], with updated presentation and notations, but otherwise identical content.

Lemma 24. *Reading an opposite index value f prevents any later acquire load on the same side from obtaining a value older than f, and any earlier acquire load from obtaining a newer value. Newer and older are relative to the modification order of* front.

Conversely, storing an owned index value f that is read by the opposite side as $\mathrm{acq}\, r^{\mathrm{f}} = f$ prevents any acquire load of the opposite index sequenced before the store from obtaining a value newer than that at the point of $\mathrm{acq}\, r^{\mathrm{f}} = f$.

The same applies symmetrically to back *instead of* front.

Formally, for all $k \geq 0$ and $k' \geq 0$. If $\mathrm{ini/rel}\, w^{\mathrm{b}} = b(P,k) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,k')}$, then:

$$\forall l < k,\ [(P,l) \textit{ is not cached}] \implies \exists l' \leq k',\ \mathrm{ini/rel}\, w^{\mathrm{f}} = f(C,l') \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{f}}_{(P,l)}$$

$$\forall l' \leq k',\ [(C,l') \textit{ is not cached}] \implies \exists l \leq k,\ \mathrm{ini/rel}\, w^{\mathrm{b}} = b(P,l) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,l')}$$

If $\mathrm{ini/rel}\, w^{\mathrm{f}} = f(C,k') \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{f}}_{(P,k)}$, then:

$$\forall l \leq k,\ [(P,l) \textit{ is not cached}] \implies \exists l' \leq k',\ \mathrm{ini/rel}\, w^{\mathrm{f}} = f(C,l') \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{f}}_{(P,l)}$$

$$\forall l' < k',\ [(C,l') \textit{ is not cached}] \implies \exists l \leq k,\ \mathrm{ini/rel}\, w^{\mathrm{b}} = b(P,l) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,l')}$$

Proof. The two sides of the lemma are symmetrical. Proof is only provided for the first scenario. We suppose that $\mathrm{ini/rel}\, w^{\mathrm{b}} = b(P,k) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,k')}$.

Let $l < k$ such that (P,l) is not cached. Since $\mathrm{acq}\, r^{\mathrm{f}}_{(P,l)} \xrightarrow{\mathrm{sb}} \mathrm{rel}\, w^{\mathrm{b}} = b(P,k) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,k')}$, we have $\mathrm{acq}\, r^{\mathrm{f}}_{(P,l)} \xrightarrow{\mathrm{hb}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,k')}$. Besides, since only the consumer writes to *front*, we know there exists l' such that $\mathrm{ini/rel}\, w^{\mathrm{f}} = f(C,l') \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{f}}_{(P,l)}$. It must be that $l' \leq k'$; otherwise, we have the following happens-before cycle:

$$\mathrm{rel}\, w^{\mathrm{f}}_{(C,l'-1)} = f(C,l') \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{f}}_{(P,l)} \xrightarrow{\mathrm{hb}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,k')} \xrightarrow{\mathrm{sb}} \mathrm{rel}\, w^{\mathrm{f}}_{(C,l'-1)} = f(C,l')$$

Let $l' \leq k'$ such that (C,l') is not cached. Since only the producer writes to *back*, we know there exists l such that $\mathrm{ini/rel}\, w^{\mathrm{b}} = b(P,l) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,l')}$. It must be that $l \leq k$. Otherwise, we have the following happens-before edge:

$$\mathrm{rel}\, w^{\mathrm{b}} = b(P,l) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,l')} \xrightarrow{\mathrm{sb}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,k')} \quad \text{hence} \quad \mathrm{rel}\, w^{\mathrm{b}} = b(P,l) \xrightarrow{\mathrm{hb}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,k')}$$

Therefore, if $\mathrm{ini/rel}\, w^{\mathrm{b}} = b(P,k) \xrightarrow{\mathrm{hb}} \mathrm{rel}\, w^{\mathrm{b}} = b(P,l)$, we have a contradiction:

$$\mathrm{ini/rel}\, w^{\mathrm{b}} = b(P,k) \not\xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,k')} = b(C,k')$$

Symmetrically if $\mathrm{ini/rel}\, w^{\mathrm{f}} = f(C,k') \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{f}}_{(P,k)} = f(P,k)$. □

From Lemma 25 to Corollary 9, we prove the following local bounds on the index values, under various hypotheses:

$$0 \leq b(T,k) - f(T,k) \leq m$$

where T is either the producer P or the consumer C, and (T,k) designates a specific instance of push or pop. We say that an instance is **bounded** if it satisfies the above predicate.

Lemma 25. *If a cached instance of push or pop is bounded, then all following operations up to and including the next non-cached instance are also bounded.*

Formally, given T either the producer P or the consumer C. For all k such that (T,k) is cached and $0 \leq b(T,k) - f(T,k) \leq m$:

$$\forall l \in \{k, \ldots, \llbracket k \rrbracket_T + 1\},\ 0 \leq b(T,l) - f(T,l) \leq m$$

Proof. If T is the producer. It follows from the definition of $\llbracket . \rrbracket_T$ that (T,l) is cached for all $l \in \{k, \ldots, \llbracket k \rrbracket_T\}$. Hence, no such (T,l) reads from *front*, and $f(T,l) = f(T,k)$. We proceed by induction on l. The hypotheses provide the base case. Suppose $0 \leq b(T,l-1) - f(T,l-1) \leq m$. $(T,l-1)$ is cached, hence there exists $n_0 \leq m < M$ such that $b(T,l) = b(T,l-1) + n_0$. Furthermore, the cached execution guarantees $(f(T,l-1) +$

$m - b(T, l-1)) \,\%\, M \geq n_0$. Since $0 \leq b(T, l-1) - f(T, l-1) \leq m$, we have $0 \leq m - (b(T, l-1) - f(T, l-1)) \leq m < M$. Hence:

$$\begin{aligned}(f(T, l-1) + m - b(T, l-1)) \,\%\, M &= (m - (b(T, l-1) - f(T, l-1))) \,\%\, M \\ &= m - (b(T, l-1) - f(T, l-1)) \geq n_0\end{aligned}$$

Since $b(T, l) = b(T, l-1) + n_0$ and $f(T, l) = f(T, l-1)$:

$$m - (b(T, l-1) - f(T, l-1)) \geq n_0 \iff m \geq b(T, l) - f(T, l)$$

If T is the consumer. It follows from the definition of $\llbracket . \rrbracket_T$ that (T, l) is cached for all $l \in \{k, \ldots, \llbracket k \rrbracket_T\}$. Hence, no such (T, l) reads from *back*, and $b(T, l) = b(T, k)$. We proceed by induction on l. The hypotheses provide the base case. Suppose $0 \leq b(T, l-1) - f(T, l-1) \leq m$. $(T, l-1)$ is cached, hence there exists $n_0 \leq m < M$ such that $f(T, l) = f(T, l-1) + n_0$. The cached execution guarantees $(b(T, l-1) - f(T, l-1)) \,\%\, M \geq n_0$. Since $0 \leq b(T, l-1) - f(T, l-1) \leq m < M$, we have:

$$(b(T, l-1) - f(T, l-1)) \,\%\, M = b(T, l-1) - f(T, l-1) \geq n_0$$

Since $f(T, l) = f(T, l-1) + n_0$ and $b(T, l) = b(T, l-1)$:

$$m \geq b(T, l-1) - f(T, l-1) \geq n_0 \iff m \geq m - n_0 \geq b(T, l) - f(T, l) \geq 0 \qquad \square$$

Lemma 26. *If an instance of push or pop reads the initial value of its opposite index, then every operation up to and including the next uncached instance is bounded.*

Formally, given push (P, k) and pop (C, k'):

$$\begin{aligned}\text{ini}\, w^{\text{b}} = 0 \xrightarrow{\text{rf}} \text{acq}\, r^{\text{b}}_{(C,k')} = 0 &\implies \forall l' \leq \llbracket k' \rrbracket_C + 1,\ 0 \leq b(C, l') - f(C, l') \leq m \\ \text{ini}\, w^{\text{f}} = 0 \xrightarrow{\text{rf}} \text{acq}\, r^{\text{f}}_{(P,k)} = 0 &\implies \forall l \leq \llbracket k \rrbracket_P + 1,\ 0 \leq b(P, l) - f(P, l) \leq m\end{aligned}$$

Proof of the first implication. Suppose $\text{ini}\, w^{\text{b}} = 0 \xrightarrow{\text{rf}} \text{acq}\, r^{\text{b}}_{(C,k')} = 0$; let us show by induction that $b(C, l') = f(C, l') = 0$ for all $l' \leq \llbracket k' \rrbracket_C$.

In the base case, if $l' = 0$, by definition, $b(C, l') = f(C, l') = 0$.

Induction case. Assume $b(C, l') = f(C, l') = 0$ and $l' \leq \llbracket k' \rrbracket_C$. Either (C, l') is cached or not. If (C, l') is not cached, then it follows from Lemma 24 that:

$$\begin{gathered}\exists l \leq 0,\ \text{ini/rel}\, w^{\text{b}} = b(P, l) \xrightarrow{\text{rf}} \text{acq}\, r^{\text{b}}_{(C,l')} = b(C, l'+1) \\ \text{hence} \quad b(C, l'+1) = b(P, 0) = 0\end{gathered}$$

- If (C, l') is failed, then $f(C, l'+1) = f(C, l')$.
- If (C, l') is not failed, then $(b(C, l'+1) - f(C, l'))\%M \geq n$, where n is the batch size argument of (C, l'); hence $n = 0$, and $f(C, l'+1) = f(C, l') + 0 = f(C, l')$. If (C, l') is cached, then $b(C, l'+1) = b(C, l') = 0$. Since it must be that $(b(C, l') - f(C, l'))\% M \geq n$, the pop must be empty: $n = 0$; thus, $f(C, l'+1) = f(C, l') + 0 = f(C, l')$.

By induction, $f(C, l' + 1) = f(C, 0) = 0$. □

Proof of the second implication. Suppose $\text{ini}\, w^{\text{f}} = 0 \xrightarrow{\text{rf}} \text{acq}\, r^{\text{f}}_{(P,k)} = 0$ and $l \leq \llbracket k \rrbracket_P$. Symmetrically to the above, we show that $f(P, l) = f(C, 0) = 0$. We prove $0 \leq b(P, l) - f(P, l) \leq m$ by induction on l:

In the base case:

$$0 \leq b(P, 0) - f(P, 0) \leq m \iff 0 \leq 0 - 0 \leq m$$

Induction case. Assume $0 \leq b(P, l) - f(P, l) \leq m$ and $l \leq \llbracket k \rrbracket_P$. We recall that $f(P, l + 1) = f(P, l) = 0$; hence $b(P, l) \leq m$. If (P, l) is failed then $b(P, l + 1) = b(P, l)$ and the property holds. If (P, l) is cached, the result follows from Lemma 25. If (P, l) is uncached, let n_0 be the batch size argument passed to (P, l). We have, for some $n_0 \leq m$:

$$\begin{aligned}(f(P, l + 1) + m - b(P, l)) \,\%\, M &= (0 + m - b(P, l)) \,\%\, M \\ &= m - b(P, l) \geq n_0\end{aligned}$$

Since $b(P, l + 1) = b(P, l) + n_0$, it follows that $b(P, l + 1) \leq m$. Moreover, $f(P, l + 1) = 0$, hence $0 \leq b(P, l + 1) - f(P, l + 1) \leq m$. □

Lemma 27. *If an instance (T, k) of push or pop reads an opposite index value written by an opposite bounded operation, then the next operation $(T, k + 1)$ on the same side as (T, k) is also bounded.*

Formally, given push (P, k) and pop (C, k'), such that $0 \leq b(C, k') - f(C, k') \leq m$ and $0 \leq b(P, k) - f(P, k) \leq m$.

$$\text{rel}\, w^{\text{b}}_{(P,k-1)} = b(P, k) \xrightarrow{\text{rf}} \text{acq}\, r^{\text{b}}_{(C,k')} = b(P, k) \implies 0 \leq b(C, k' + 1) - f(C, k' + 1) \leq m$$
$$\text{rel}\, w^{\text{f}}_{(C,k'-1)} = f(C, k') \xrightarrow{\text{rf}} \text{acq}\, r^{\text{f}}_{(P,k)} = f(C, k') \implies 0 \leq b(P, k + 1) - f(P, k + 1) \leq m$$

Proof. The two sides of the lemma being symmetrical, we only provide the proof for the first scenario. Suppose $\text{rel}\, w^{\text{b}}_{(P,k-1)} = b(P, k) \xrightarrow{\text{rf}} \text{acq}\, r^{\text{b}}_{(C,k')} = b(P, k)$. We have $b(C, k' + 1) = b(P, k)$ and $f(C, k' + 1) = f(C, k') + n_0$ for some $n_0 \geq 0$ (if (C, k') is failed, $n_0 = 0$).

Since only the consumer writes to *front*, there is k'_0 such that $f(P, k) = f(C, k'_0)$. It follows from Lemma 24 that $k'_0 \leq k'$. Hence, $f(P, k) = f(C, k'_0) \leq f(C, k')$ and:

$$b(C, k' + 1) - f(C, k' + 1) = b(P, k) - (f(C, k') + n_0) \leq b(P, k) - f(P, k) \leq m$$

Consequently, $b(C, k' + 1) - f(C, k') \leq m + n_0 < M$. Furthermore, $b(C, k') \leq b(C, k' + 1)$, hence:

$$0 \leq b(C, k') - f(C, k') \leq b(C, k' + 1) - f(C, k') < M$$

If (C, k') is failed, then $n_0 = 0$; thus, $0 \leq b(C, k') - f(C, k') \leq b(C, k'+1) - f(C, k'+1)$. Otherwise, (C, k') satisfies $(b(C, k'+1) - f(C, k')) \ \% \ M = b(C, k'+1) - f(C, k') \geq n_0$. Thus:

$$b(C, k'+1) - f(C, k') \geq n_0 \iff b(C, k'+1) - f(C, k'+1) \geq 0$$

Symmetrically if $\mathrm{rel}\, w^{\mathrm{f}}_{(C,k'-1)} = f(C, k') \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{f}}_{(P,k)} = f(C, k')$. □

Lemma 28. *If an instance of push or pop reads a opposite index value from a release store, then every operation up to and including the next uncached instance is bounded.*

Formally, given push (P, k) and pop (C, k'). If (C, k') is not cached, then the following holds:

$$\forall l' \in \{k'+1, \ldots, \llbracket k' \rrbracket_C + 1\},\ 0 \leq b(C, l') - f(C, l') \leq m$$

If (P, k) is not cached, then the following holds:

$$\forall l \in \{k+1, \ldots, \llbracket k \rrbracket_P + 1\},\ 0 \leq b(P, l) - f(P, l) \leq m$$

Proof. The two sides of the lemma being symmetrical, we only provide the proof for the (C, k') scenario. (C, k') is not cached; hence $\mathrm{acq}\, r^{\mathrm{b}}_{(C,k')} = b(C, k')$ reads from either the initial write to *back* or there is k_0 such that $\mathrm{rel}\, w^{\mathrm{b}} = b(P, k_0) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,k')} = b(C, k')$.

If (C, k') reads from the initial write, then it follows from Lemma 26 that the property holds. Otherwise, (C, k') reads from a release store, and we know that $\mathrm{rel}\, w^{\mathrm{b}}_{(P,k_0-1)} = b(P, k_0) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,k')} = b(P, k_0)$. We proceed by induction on $\max(k_0, k'+1)$.

In the base case, Lemma 26 asserts that the first push $(P, 0)$ can only read the initial value of *front*, and that the property holds for that pair. The first pop $(C, 0)$ can either read from the initial write to *back* or from a push (P, k_0). If it reads from the initial write, then the property holds from Lemma 26. Otherwise, $(C, 0)$ reads from $\mathrm{ini/rel}\, w^{\mathrm{b}} = b(P, k_0)$, and it follows from Lemma 24 that $f(P, k_0)$ is the initial value of *front*, since no non-zero write to *front* exists before the first pop. Hence, Lemma 26 implies that $0 \leq b(P, k_0) - f(P, k_0) = b(C, 0) - f(C, 0) \leq m$.

Let us prove the induction step. Assume the property holds for all (P, l) and (C, l'), such that $\max(l, l'+1) < \max(k_0, k'+1)$, and $\mathrm{ini/rel}\, w^{\mathrm{b}} = b(P, l) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_{(C,l')} = b(P, l)$ or $\mathrm{ini/rel}\, w^{\mathrm{f}} = f(C, l) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{f}}_{(P,l')} = f(C, l)$.

Let us consider $\mathrm{ini/rel}\, w^{\mathrm{f}} = f(C, k'_0) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{f}}_{\llbracket P, k_0-1 \rrbracket} = f(C, k'_0)$ for some k'_0. It follows from Lemma 24 that $k'_0 \leq k'$. Two cases:

- If $k_0 \leq k'$, then $\max(k_0, k'+1) = k'+1$. Since $k'_0 < k'+1$ and $\llbracket k_0 - 1 \rrbracket_P \leq k_0 - 1 < k'+1$, we have $\max(k'_0, \llbracket k_0 - 1 \rrbracket_P + 1) < k'+1 = \max(k_0, k'+1)$. By induction, the property is true for $(k'_0, \llbracket k_0 - 1 \rrbracket_P + 1)$, and, from Lemma 25, we have $0 \leq b(P, k_0) - f(P, k_0) \leq m$.

Similarly, the property holds for $\max(k_{-1}, \llbracket k'-1 \rrbracket_C + 1) < \max(k_0, k'+1)$ (for some matching k_{-1}), and $0 \le b(C,k') - f(C,k') \le m$. Hence, Lemma 27 concludes that $0 \le b(C,k'+1) - f(C,k'+1) \le m$. Lemma 25 extends the bounds to all cached direct successors.

- Otherwise, $k_0 > k'$. There exists k_1 such that we have: $\text{ini/rel}\, w^{\text{b}} = b(P,k_1) \xrightarrow{\text{rf}} \text{acq}\, r^{\text{b}}_{\llbracket C, k'_0 - 1 \rrbracket} = b(C,k_1)$. Moreover, it follows from Lemma 24 that $k_1 \le k_0 - 1$. Since the property holds for all $\max(l, l'+1) < \max(k_0, k'+1)$, it does in particular for all $\max(l, l'+1) < \max(k'_0, \llbracket k_0 - 1 \rrbracket_P + 1)$. By symmetry with the above case, we get $0 \le b(P,k_0) - f(P,k_0) \le m$.

  It remains to be shown that $0 \le b(C,k') - f(C,k') \le m$. Either there is $k'_1 \le \llbracket k'-1 \rrbracket_C$ such that $\text{ini/rel}\, w^{\text{b}} = b(C,k_1) \not\xrightarrow{\text{rf}} \text{acq}\, r^{\text{b}}_{\llbracket C, k'_1 \rrbracket} = b(C,k_1)$, or there is no such k'_1.

  If there is no such k'_1, then consider the first pop; let $k'_1 = 0$. We have a base case: $0 \le b(C,k'_1) - f(C,k'_1) \le m$. If k'_1 exists, then $\max(k_{-1}, k'_1) < \max(k_0, k'+1)$ (for some matching k_{-1}) and, by induction, $0 \le b(C,k'_1) - f(C,k'_1) \le m$.

  In both cases, $0 \le b(C,k'_1) - f(C,k'_1) \le m$, and by repeatedly applying Lemma 25 and Lemma 27, we get $0 \le b(C,k'+1) - f(C,k'+1) \le m$. Lemma 25 extends the bounds to all cached direct successors.

□

Corollary 9. *All instances of push or pop are bounded. In other words, for T either the producer P or the consumer C, and all $k \ge 0$, we have:* $0 \le b(T,k) - f(T,k) \le m$.

Proof. Consider the previous push (resp. pop) that is not cached $\llbracket T, k-1 \rrbracket$. If there is none, Lemma 26 concludes. If there is such an operation, then Lemma 28 applies to $\llbracket T, k-1 \rrbracket$ and the result holds for $k \in \{\llbracket k-1 \rrbracket_T + 1, \dots, \lceil\!\lceil k-1 \rceil\!\rceil_T + 1\}$. □

Corollary 10. *All accesses—both loads or stores—to the data buffer take place at an index within the local bounds previously established.*

Formally, given a push (P,k) and a store $w^{[i]}$ in (P,k):

$$0 \le b(P,k) \le i < f(P,k) + m$$

And given a pop (C,k') and a load $r^{[j]}$ in (C,k'):

$$0 \le f(C,k') \le j < b(C,k')$$

Proof. For such a store to occur, (P,k) must not be failed; assume (P,k) stores n_0 elements. From Corollary 9, we have:

$$\begin{aligned} & 0 \le b(P,k) - f(P,k) \le m \\ & \quad \implies 0 \le (f(P,k) + m - b(P,k)) = m - (b(P,k) - f(P,k)) \le m \\ & \quad \implies 0 \le n_0 \le (f(P,k) + m - b(P,k)) \,\%\, M = f(P,k) + m - b(P,k) \le m \end{aligned}$$

Therefore, $0 \leq b(T,k) \leq i < b(C,k) + n_0 \leq f(P,k) + m$.

Similarly, for such a load to occur, (C,k') must not be failed; assume (C,k') reads n_0 elements. Corollary 9 gives:

$$0 \leq n_0 \leq (b(C,k') - f(C,k')) \,\%\, M = b(C,k') - f(C,k') \leq m$$

Hence, $0 \leq f(T,k) \leq j < f(C,k') + n_0 \leq b(C,k')$. □

The remaining lemmas and theorems pertain to the data transferred through the single-producer single-consumer queue. We recall that all accesses to the data buffer are made by the queue code alone. Consequently, any load (resp. store) from the data buffer is implicitly assumed to take place during a pop (resp. push).

Lemma 29. *Reading from the data buffer yields a well-defined value, written by a corresponding store.*

In other words, given a load $r^{[j]}$ from an instance of pop: $\exists w^{[i]},\ w^{[i]} \xrightarrow{\text{rf}} r^{[j]}$.

Proof. Sequentially, a store $\text{rel}\, w^{\text{b}} = n$ in a push is always preceded by writes $w^{[i]}$ for all $i < n$. Let (C,k') be the pop $r^{[j]}$ belongs to. It follows from Corollary 10 that $0 \leq j < b(C,k')$; thus we know that $\text{ini}\, w^{\text{b}} = 0 \not\xrightarrow{\text{rf}} b(C,k')$. There is a push (P,l) that stores $\text{rel}\, w^{\text{b}} = b(C,k')$, such that:

$$w^{[j]} \xrightarrow{\text{sb}} \text{rel}\, w^{\text{b}}_{(P,l)} = b(C,k') \xrightarrow{\text{rf}} \text{acq}\, r^{\text{b}}_{\llbracket C,k' \rrbracket} = b(C,k') \xrightarrow{\text{sb}} r^{[j]}$$

Consequently, $\text{ini}\, w^{[i]} \not\xrightarrow{\text{rf}} r^{[j]}$; the value read must come from some $w^{[i]}$. □

Lemma 30. *A load from the data buffer reads exactly the value written by a store at the same extended index (in $\mathbf{N}$). In other words, if $w^{[i]} \xrightarrow{\text{rf}} r^{[j]}$, then $i = j$.*

Proof. Suppose $w^{[i]} \xrightarrow{\text{rf}} r^{[j]}$. Let (P,k) be the push $w^{[i]}$ belongs to and (C,k') the pop $r^{[j]}$ belongs to. Corollary 10 asserts that $j < b(C,k')$. We have two cases:

- If $j < i+1$, then there exists $q > 0$ such that $j = i - qm$. Therefore, on the one hand $f(C,k') \leq j \leq i - m$. On the other hand, since (P,k) stores at i, $f(P,k) \geq i - m + 1$. Hence, $f(C,k') < f(P,k)$ and there is a pop $l' \geq k'$ that writes the value $f(P,k)$. We have a happens-before cycle. Impossible.

- Otherwise, $j \geq i$. Suppose $j > i$; then there exists $q > 0$ such that $j = i + qm$. We have $b(C,k') > j \geq i + m$. Hence there must be a push (P,l) that stores $\text{rel}\, w^{\text{b}} = b(C,k')$, such that:

  $$w^{[i+m]} \xrightarrow{\text{sb}} \text{rel}\, w^{\text{b}}_{(P,l)} = b(C,k') \xrightarrow{\text{rf}} \text{acq}\, r^{\text{b}}_{\llbracket C,k' \rrbracket} = b(C,k')$$

  Furthermore, $w^{[i]} \xrightarrow{\text{hb}} w^{[i+m]}$, thus it cannot be that $w^{[i]} \xrightarrow{\text{rf}} r^{[j]}$. Therefore, it must be that $i = j$. □

Lemma 31. *All stores to the data buffer at some index i happen before any load at an index $j > i$.*

Formally, given a store $w^{[i]}$ and a load $r^{[j]}$, we have the following implication:

$$i \leq j \implies w^{[i]} \xrightarrow{\text{hb}} r^{[j]}$$

Proof. Suppose $i \leq j$. It follows from Corollary 10 that $b(C, k') > j \geq i$. Hence there must be a store $\text{rel}\, w^{\text{b}} = b(C, k')$ in the producer sequenced after $w^{[i]}$, such that:

$$w^{[i]} \xrightarrow{\text{sb}} \text{rel}\, w^{\text{b}} = b(C, k') \xrightarrow{\text{rf}} \text{acq}\, r^{\text{b}} = b(C, k') \xrightarrow{\text{sb}} r^{[j]} \qquad \square$$

We now move on to prove the theorems formulated in Section 4.2.2.3.

Proof of Theorem 3. Suppose there are two such loads, $r^{[j]}$ and $r^{[j']}$ with $j \neq j'$:

$$w^{[i]} \xrightarrow{\text{rf}} r^{[j]} \quad \text{and} \quad w^{[i]} \xrightarrow{\text{rf}} r^{[j']}$$

It follows from Lemma 30 that $i = j = j'$. Impossible. □

Proof of Theorem 4. Sequentially, a store $\text{rel}\, w^{\text{f}} = m$ in a pop is always preceded by reads $r^{[j]}$ for all $j < m$. Hence, if $\sum_{k' \in \text{NFC}} n_{k'} \geq i$, then there is a pop that stores $\text{rel}\, w^{\text{f}} = m$ for some $m > i$. Hence there exists a load $r^{[i]}$; it follows from Lemma 29 that we have a store $w^{[i']} \xrightarrow{\text{rf}} r^{[i]}$, and from Lemma 30 that $i' = i$. □

Proof of Theorem 5. Suppose we have $w^{[j]}$. The store belongs to the producer side, as does $w^{[i]}$; hence the two are sequentially ordered.

Suppose we have $r^{[j]}$ in pop (C, k'). If $i \leq j$, then by Lemma 31, $w^{[i]} \xrightarrow{\text{hb}} r^{[j]}$. Otherwise, $i > j$. Let (P, k) be the push to which the store belongs. There exists $q > 0$ such that $i = qm + j$. Furthermore, it follows from Corollary 10 that $i < f(P, k) + m$. Thus:

$$i < f(P, k) + m \iff i - m < f(P, k) \iff qm + j - m < f(P, k)$$

Hence, $f(C, k') \leq j < f(P, k)$. Therefore, there must be a store $\text{rel}\, w^{\text{f}} = f(P, k)$, such that:

$$r^{[j]} \xrightarrow{\text{sb}} \text{rel}\, w^{\text{f}} = f(P, k) \xrightarrow{\text{rf}} \text{acq}\, r^{\text{f}} = f(P, k) \xrightarrow{\text{sb}} w^{[i]} \qquad \square$$

A.2 Proof of the Chase–Lev work-stealing deque

This section adapts the work from [Lê et al., 2013] to the C11 memory model. The overall layout stays identical, although several individual proofs have been modified—usually simplified—substantially.

A.2.1 Significant reads and writes

Lemma 32. *Given a write $w^{a[i]}$ and a read $r^{a'[j]}$, $i \neq j \implies w^{a[i]} \overset{\text{rf}}{\not\longrightarrow} r^{a'[j]}$.*

Proof. If the addresses of the underlying arrays differ, then the memory locations read and written are distinct and there can be no reads-from relation.

Otherwise, since old arrays are never reused, the addresses are the same and $i \equiv j \mod n_a$. We know that $r^{a'[j]}$ belongs to a successful instance of *take*, *give* (with resizing), or *steal*. Let X be that instance.

Let P be the instance of *give* to which $w^{a[i]}$ belongs. In P, we have the following action structure:

$$\begin{aligned} &\text{acq}\, r_P^t = t_P \xrightarrow{\text{sb}} \text{rlx}\, w^{a[i]} \xrightarrow{\text{sb}} \text{rel}\, f \xrightarrow{\text{sb}} \text{rlx}\, w^b = b_P + 1 \\ &\text{where} \quad t_P \leq i \leq b_P \wedge b_P - t_P < n_a - 1 \end{aligned}$$

Let us assume $i \neq j \wedge w^{a[i]} \xrightarrow{\text{rf}} r^{a'[j]}$ and show it is indeed impossible.

Assume X is a successful instance of *take* or *give*. Since X and P belong to the same thread, P must be sequenced before X. Most notably $w^{a[i]} \xrightarrow{\text{hb}} r^{a'[j]}$.

- If $j < i$, then $j \leq i - n_a$ (due to modulo access). However, the following must hold in P:

  $$\begin{aligned} &t_P \leq i \leq b_P \wedge b_P - t_P < n_a - 1 \\ \text{hence} \quad &j < i - n_a + 1 \leq b_P - n_a + 1 < t_P \end{aligned}$$

  Furthermore, if X is a *take* operation, $r^{a'[j]}$ reads the last element of the deque, and $j = b_X - 1 \geq t_X$ (where b_X and t_X are the indices read in X); if X is a *give* operation, $r^{a'[j]}$ results from a copy operation of the resizing code, hence $j \geq t_X$. Since X occurs after P in program order and *top* is monotonically increasing, $r_P^t = t_P \xrightarrow{\text{hb}} r_X^t = t_X$ and $j < t_P \leq t_X \leq j$. Impossible.

- If $i < j$, then, since $j \geq i + n_a \geq b_P$, *bottom* must increase from b_P to $j+1$ between the write in P and the read in X. Hence, there must be an instance P' of *give* between P and X (in program order) that increments *bottom* to $j+1$. Indeed, the only writes that increase the value of *bottom* occur in *give* and *take*; and the effect of *take* as a whole never increases the value of *bottom* since it first decrements the variable. We have:

  $$w_P^{a[i]} \xrightarrow{\text{mo}} w_{P'}^{a[j]} \xrightarrow{\text{hb}} r_X^{a'[j]}$$

  Thus, $w_P^{a[i]} \overset{\text{rf}}{\not\longrightarrow} r_X^{a'[j]}$ by coherence.

Now, assume X is a successful instance of *steal*. We have the following execution graph for X:

$$\begin{aligned} &\text{rlx}\, r_X^{\text{t}} = t_X = j \xrightarrow{\text{sb}} \text{sc}\, f \xrightarrow{\text{sb}} \text{acq}\, r^{\text{b}} = b_X \\ &\qquad \xrightarrow{\text{sb}} \text{acq}\, r^{\text{a}} = a \xrightarrow{\text{sb}} \text{rlx}\, r^{a'[j]} \xrightarrow{\text{sb}} \text{sc}\, s^t = t_X \to t_X + 1 \end{aligned}$$

- If $j < i$, then $j \le i - n_a$. As above, the following must hold in P:

  $$j < i - n_a + 1 \le b_P - n_a + 1 < t_P$$

  Hence $t_X = j < t_P$. Since *top* increases monotonically, it must be that:

  $$\begin{aligned} r_X^{a'[j]} &\xrightarrow{\text{sb}} s^t = t_X \to t_X + 1 \\ &\xrightarrow{\text{rf}} \text{acq}\, r^t \xrightarrow{\text{sb}} \text{sc}\, f' \xrightarrow{\text{sb}} s^t \xrightarrow{\text{rf}} \cdots \xrightarrow{\text{rf}} s^t = t_p - 1 \to t_P \\ &\xrightarrow{\text{rf}} \text{acq}\, r_P^t = t_P \xrightarrow{\text{sb}} w^{a[i]} \end{aligned}$$

  The compare-and-swap operations are sequentially consistent, thus imply release–acquire, and $r_X^{a'[j]} \xrightarrow{\text{hb}} w^{a[i]} \xrightarrow{\text{rf}} r_X^{a'[j]}$. Impossible.

- If $i < j$, then $j \ge i + n_a$, and there must be an instance P' of *give* such that $W[P']b = j + 1$ $(\xrightarrow{\text{sb}} w^b = b_X) \xrightarrow{\text{rf}} r_X^b = b_X$, so that index j be accessible in X. P' cannot occur before P in program order because, as above, we would have $t_{P'} \le t_P \le i$ on the one hand, and $i \le j - n_a < t_{P'}$ on the other hand. The underlying array also monotonically increases in size, so the inequality still holds if the sizes in P and P' differ. Hence P' occurs after P. Furthermore, there exists $w_{P'}^{a''[j]}$ in P'.

  If a in P and a'' in P' refer to different arrays, then a resize operation R must precede P', such that:

  $$\begin{aligned} w^{\text{a}} = a &\xrightarrow{\text{sb}} r_P^{\text{a}} = a \xrightarrow{\text{sb}} w_R^{\text{a}} = a'' \xrightarrow{\text{sb}} w_{P'}^{a''[j]} \\ &\xrightarrow{\text{sb}} \text{rel}\, w^{\text{b}} = j + 1 \ (\xrightarrow{\text{sb}} w^{\text{b}} = b_X) \\ &\xrightarrow{\text{rf}} \text{acq}\, r_X^{\text{b}} = b_X \xrightarrow{\text{sb}} \text{acq}\, r^{\text{a}} = a = a' \xrightarrow{\text{sb}} r^{a'[j]} \end{aligned}$$

  $w^{\text{b}} = b_X$ is either release or in the release sequence of a previous release write on *bottom* in the same thread (since only the owner modifies *bottom*). Thus, $w^{\text{a}} = a \xrightarrow{\text{mo}} w_R^{\text{a}} = a'' \xrightarrow{\text{hb}} r_X^{\text{a}} = a'$, and therefore $w^{\text{a}} = a \not\xrightarrow{\text{rf}} r_X^{\text{a}} = a'$. Since arrays are never reused, $a' \neq a$. Impossible.

  Otherwise, a and a'' refer to the same array, hence $a[i]$ and $a''[j]$ refer to the same location, and we get:

  $$\begin{aligned} w_P^{a[i]} &\xrightarrow{\text{sb}} w_{P'}^{a''[j]} \xrightarrow{\text{sb}} \text{rel}\, w^{\text{b}} = j + 1 (\xrightarrow{\text{sb}} w^{\text{b}} = b_X) \\ &\xrightarrow{\text{rf}} \text{acq}\, r_X^{\text{b}} = b_X \xrightarrow{\text{sb}} r_X^{a'[j]} \end{aligned}$$

  Thus, $w_P^{a[i]} \xrightarrow{\text{mo}} w_{P'}^{a''[i]} \xrightarrow{\text{hb}} r_X^{a'[j]}$, and therefore $w_P^{a[i]} \not\xrightarrow{\text{rf}} r_X^{a'[j]}$. Impossible. □

Corollary 11. *Given a significant write $w^{a[i]} = x_i(v)$ and a significant read $r^{a'[j]}$:*

$$i \neq j \implies w^{a[i]} \not\xrightarrow{\text{rff}} r^{a[j]}$$

Proof. If $i \neq j$, we know that $w^{a[i]} \overset{\text{rf}}{\not\longrightarrow} r^{a[j]}$. Furthermore, all copies, which happen during a resize operation, copy from and to the same index. Since there are less copies than the size of the expanded array, there can be no two copies writing to the same memory location in the new array. Hence, there can be no sequence of copies from $w^{a[i]}$ to $r^{a[j]}$. □

Lemma 33. *Given a significant write $w = x_i(u)$ and a significant read $r^{a[i]} = x_i(v)$:*

$$w = x_i(u) \xrightarrow{\text{hb}} r^{\text{a}} = a \xrightarrow{\text{sb}} r^{a[i]} = x_i(v) \implies u \leq v$$

Proof. Suppose $v < u$. We define $w^{a[i]}_{W'} = x_i(v)$ as follows.

If $v = 0$, $x_i(v) = \bot$ is an undefined value; let $w^{a[i]}_{W'} = x_i(0) \xrightarrow{\text{rf}} r^{a[i]} = x_i(v)$ be the initialization of $a[i]$. $w^{a[i]}_{W'}$ comes before w in program order.

Otherwise, $0 < v < u$. Let $w' = x_i(v)$ be the significant write such that $w' \xrightarrow{\text{rff}} r^{a[i]} = x_i(v)$. In other words, there exists a sequence of copies carrying the value $x_i(v)$ to $r^{a[i]} = x_i(v)$. That sequence ends with a write $w^{a[i]}_{W'} = x_i(v) \xrightarrow{\text{rf}} r^{a[i]} = x_i(v)$. By the definition of x_i, since $v < u$, it must be that $w' \xrightarrow{\text{sb}} w$. According to the sequential semantics of resizing, $w^{a[i]}_{W'} \xrightarrow{\text{sb}} w$ too, since otherwise $w^{a[i]}_{W'}$ could not copy from w' (either overwritten or in a different array).

We have two cases: either w and $r^{a[i]} = x_i(v)$ refer to the same memory location or they do not.

- Assume that they refer to the same memory location $a[i]$. Then we have:

  $$w^{a[i]}_{W'} = x_i(v) \xrightarrow{\text{mo}} w = x_i(u) \xrightarrow{\text{hb}} r^{a[i]} = x_i(v)$$

  Yet $w^{a[i]}_{W'} \xrightarrow{\text{rf}} r^{a[i]} = x_i(v)$ by definition. Impossible.

- Conversely, assume that they do not refer to the same memory location. Then there must be a resize operation between $w^{a[i]}_{W'}$ and w:

  $$w^{\text{a}} = a \xrightarrow{\text{sb}} w^{a[i]}_{W'} = x_i(v) \xrightarrow{\text{sb}} w^{\text{a}} = a' \xrightarrow{\text{sb}} w = x_i(u) \xrightarrow{\text{hb}} r^{\text{a}} = a \xrightarrow{\text{sb}} r^{a[i]} = x_i(v)$$

  Therefore, $w^{\text{a}} = a \overset{\text{rf}}{\not\longrightarrow} r^{\text{a}} = a$. Since there is only one write that gives the value a to *array*, we have a contradiction.

□

Lemma 34. *All load operations $r^{a[i]} = x$ in successful instances of* take *or* steal *read initialized values: $x = x_i(v)$ for some $v > 0$.*

Proof. Let X be the successful instance of *take* or *steal* that contains $r^{a[i]} = x$.

We assume that only *give* writes to *array*. Suppose $x = \bot$, then it can only be an undefined value from the uninitialized array, which is not copied during resizing. Otherwise, the release store on *array* ensures that copies are seen by any thread that reads the new *array* pointer. Such a copy corresponds to some significant write.

Therefore, $a[i]$ is not affected by copying. Then it must be one of the new slots allocated by the resizing, and its initial value is $x_i(0)$. Let R be the resizing *give* operation that allocates the array x. There exists a $x_i(u)$ such that:

$$\begin{aligned} \mathrm{ini}\, w^{a[i]} = \bot \xrightarrow{\mathrm{mo}} \mathrm{rlx}\, w_R^{a[i]} = x_i(u) \xrightarrow{\mathrm{sb}} \mathrm{rel}\, w^{\mathrm{a}} = a \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r_X^{\mathrm{a}} = a \xrightarrow{\mathrm{sb}} r^{a[i],\xi} \quad & \text{if in } \textit{steal} \\ \cdots \xrightarrow{\mathrm{sb}} w^{\mathrm{a}} = a \xrightarrow{\mathrm{sb}} r_X^{\mathrm{a}} = a \xrightarrow{\mathrm{sb}} r^{a[i],\xi} \quad & \text{if in } \textit{take} \end{aligned}$$

Thus, $\mathrm{ini}\, w^{a[i]} = \bot \overset{\mathrm{rf}}{\not\longrightarrow} r^{a[i]} = x$. Impossible. □

A.2.2 Uniqueness of significant reads

The results from the previous section establish that two significant reads at different indexes cannot retrieve the same element $x_i(v)$. The only possible cause of duplicate significant reads is thus reduced to the case where the reads access the same index i.

Lemma 35. *Given S and S' distinct successful instances of* steal*,*

$$\forall r = x_i(v) \in S,\ \forall r' = x_{i'}(v') \in S',\ i \neq i'$$

Proof. All writes to *top* atomically increment it (by atomicity of compare-and-swap). Hence two successful *steal* operations cannot write (thus read) the same value of *top*. Elements fetched in a *steal* operation access the index given by the value of the *top* variable. Hence $r^{\mathrm{t}} = i \in S$ and $r^{\mathrm{t}} = i' \in S'$ imply $i \neq i'$. □

Lemma 36. *Given T a successful instance of* take *and P an instance of* give*. If P comes after T in program order, then:*

$$\forall r = x_i(v) \in T,\ \forall w = x_j(u) \in P,\ i \neq j \vee v \neq u$$

Proof. Assume $i = j \wedge v = u$. We have $r^{\mathrm{a}} \xrightarrow{\mathrm{sb}} r \xrightarrow{\mathrm{sb}} w$; therefore, from the contraposition of Lemma 33, it follows that $u > v$. We have a contradiction. □

Lemma 37. *Given T and T' distinct successful instances of* take*:*

$$\forall r = x_i(v) \in T,\ \forall r' = x_{i'}(v') \in T',\ i \neq i' \vee v \neq v'$$

Proof. We have the following action structures:

$$\begin{aligned} r^{\mathrm{b}}_T = b(n) &\xrightarrow{\mathrm{sb}} w^{\mathrm{b}} = b(n) - 1 \xrightarrow{\mathrm{sb}} r^{\mathrm{t}} = t \xrightarrow{\mathrm{sb}} r = x_{b(n)-1}(v) \xrightarrow{\mathrm{sb}} \cdots \\ r^{\mathrm{b}}_{T'} = b(n') &\xrightarrow{\mathrm{sb}} w^{\mathrm{b}} = b(n') - 1 \xrightarrow{\mathrm{sb}} r^{\mathrm{t}} = t' \xrightarrow{\mathrm{sb}} r' = x_{b(n')-1}(v') \xrightarrow{\mathrm{sb}} \cdots \end{aligned}$$

Hence, $b(n) - 1 = i$ and $b(n') - 1 = i'$.

Since all instances of *take* occur in the same worker thread, we have:

$$w^{\mathrm{b}}_T = b(n) - 1 \xrightarrow{\mathrm{sb}} r^{\mathrm{b}}_{T'} = b(n') \vee w^{\mathrm{b}}_{T'} = b(n') - 1 \xrightarrow{\mathrm{sb}} r^{\mathrm{b}}_T = b(n)$$

Let us assume the first case as well as $i = i' \wedge v = v'$ and show it is impossible, the other case being symmetrical. We have $b(n)-1 = i = i' = b(n')-1$, and $w^{\mathrm{b}}_T = i \xrightarrow{\mathrm{sb}} r^{\mathrm{b}}_{T'} = i+1$.

Hence *bottom* must increase from i to $i+1$ between n and n': there exists an instance P of *give* that writes $w^{\mathrm{b}} = b(k) \xrightarrow{\mathrm{sb}} r^{\mathrm{b}}_{T'} = i + 1$, such that $n < k \leq n'$, $b(k-1) = i$ and $b(k) = i + 1$ (as noted above, *take* as a whole does not increase the value of *bottom*). We get the following graph:

$$r^{\mathrm{b}} = i \xrightarrow{\mathrm{sb}} w = x_i(u) \xrightarrow{\mathrm{sb}} w^{\mathrm{b}} = b(k) = i + 1 \xrightarrow{\mathrm{sb}} r^{\mathrm{b}}_{T'} = b(n') \xrightarrow{\mathrm{sb}} r' = x_{i=i'}v'$$

It follows from Lemma 33 that $u \leq v'$ and from Lemma 36 that $v < u$. Impossible. □

Lemma 38. *Given T a successful instance of* take *and S a successful instance of* steal*:*

$$\forall r = x_i(v) \in T,\ \forall r' = x_{i'}(v') \in S,\ i \neq i' \vee v \neq v'$$

Proof. We have the following execution graphs:

$$\begin{aligned} \mathrm{rlx}\, r^{\mathrm{b}}_T = b(n) &\xrightarrow{\mathrm{sb}} \mathrm{rlx}\, w^{\mathrm{b}} = b(n) - 1 \xrightarrow{\mathrm{sb}} \mathrm{sc}\, f \\ &\xrightarrow{\mathrm{sb}} \mathrm{rlx}\, r^{\mathrm{t}} = t(m) \xrightarrow{\mathrm{sb}} \mathrm{rlx}\, r = x_{i=b(n)-1}(v) \xrightarrow{\mathrm{sb}} \cdots \\ \mathrm{rlx}\, r^{\mathrm{t}}_S = t(m') &\xrightarrow{\mathrm{sb}} \mathrm{sc}\, f'\, \mathrm{acq}\, \mathrm{acq}\, r^{\mathrm{b}} = b(n') \\ &\xrightarrow{\mathrm{sb}} \mathrm{rlx}\, r' = x_{i'=t(m')}(v') \xrightarrow{\mathrm{sb}} \mathrm{sc}\, s^{\mathrm{t}} = t(m') \to t(m') + 1 \end{aligned}$$

Let us assume $i = i' \wedge v = v'$. Then $t(m') = i' = i = b(n) - 1$. For S to succeed, we must have $t(m') < b(n')$. Hence, $b(n) \leq b(n')$.

Also, for T to succeed, we must have $t(m) < b(n)$. Two cases:

- If $b(n) = t(m) + 1$, then a successful compare-and-swap occurs in T. Moreover, $b(n) = t(m)+1$ implies $t(m')+1 = b(n) = t(m)+1$, hence $t(m') = t(m)$. Impossible, since *top* is monotonically increasing and S must also contain a successful compare-and-swap with the same value of *top*.

- If $b(n) > t(m) + 1$, then no compare-and-swap occurs in T and $t(m') = b(n) - 1 \geq t(m) + 1 > t(m)$. Since *top* monotonically increases, there must be two writes $s^{\mathrm{t}}_1 = t(m) - 1 \to t(m) \xrightarrow{\mathrm{mo}} s^{\mathrm{t}}_2 = t(m') - 1 \to t(m')$ such that:

  $$\mathrm{rel}\, s^{\mathrm{t}}_1 \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{t}}_T = t(m) \wedge \mathrm{rel}\, s^{\mathrm{t}}_2 \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{t}}_S = t(m') \xrightarrow{\mathrm{sb}} r^{b(n')}$$

  The fences f and f' are totally ordered by sequential consistency. Suppose $f' \xrightarrow{\mathrm{sc}} f$. Then, we have:

  $$\mathrm{sc}\, s^{\mathrm{t}}_2 \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{t}}_S = t(m') \xrightarrow{\mathrm{sb}} \mathrm{sc}\, f' \xrightarrow{\mathrm{sc}} \mathrm{sc}\, f$$
  $$\text{hence} \quad s^{\mathrm{t}}_1 \xrightarrow{\mathrm{mo}} s^{\mathrm{t}}_2 \xrightarrow{\mathrm{sc}} f' \xrightarrow{\mathrm{sc}} f \xrightarrow{\mathrm{sb}} r^{\mathrm{t}}_T$$

  Thus, r^{t}_T cannot read from s^{t}_1, which occurs before s^{t}_2 in modification order. Impossible.

  Therefore $w^{\mathrm{b}}_T = b(n+1) = b(n) - 1 \xrightarrow{\mathrm{sb}} f \xrightarrow{\mathrm{sc}} f' \xrightarrow{\mathrm{sb}} r^{\mathrm{b}}_S = b(n')$, and $n' \geq n + 1$; i.e., $w^{\mathrm{b}} = b(n')$, which is read by $r^{\mathrm{b}}_S = b(n')$, must not come before $w^{\mathrm{b}}_T = b(n) - 1$. Consequently, *bottom* must increase from $b(n) - 1 = i$ to $b(n')$ between $n + 1$ and n'. Since T does not increment the value of *bottom* (execution without compare-and-swap), there must be an instance P of *give* after T that writes $w^{\mathrm{b}}_P = b(k) \xrightarrow{\mathrm{sb}} w^{\mathrm{b}} = b(n') \xrightarrow{\mathrm{rf}} r^{\mathrm{b}}_S = b(n')$, such that $n < k \leq n' \wedge b(k-1) = i \wedge b(k) = i + 1$.

  We get the following execution graph:

  $$\begin{aligned} w^{a[i]}_P = x_i(u) \xrightarrow{\mathrm{sb}} \mathrm{rel}\, w^{\mathrm{b}} = i + 1\ (\xrightarrow{\mathrm{sb}} w^{\mathrm{b}} = b(n')) \\ \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_S = b(n') \xrightarrow{\mathrm{sb}} r^{\mathrm{a}} = a \xrightarrow{\mathrm{sb}} r' = x_{i'}(v') \end{aligned}$$

  As before, $w^{\mathrm{b}} = b(n')$ is either release or in the release sequence of a previous release store in *give*. Thus it follows from Lemma 33 that $u \leq v'$, and from Lemma 36 that $v < u \leq v'$. We have a contradiction. □

Theorem 6 follows directly from Lemmas 35, 37 and 38.

A.2.3 Existence of significant reads

Let P_{F} be the last instance of *give* in the worker thread, in program order, with $w^{\mathrm{b}}_{P_{\mathrm{F}}} = b(n_{\mathrm{F}})$ and $r^{\mathrm{t}}_{P_{\mathrm{F}}} = t(m_{\mathrm{F}})$. We say that an instance X of *take* or *steal* is **trailing** if $\exists n \geq n_{\mathrm{F}},\ r^{\mathrm{b}} = b(n) \in X$.

Lemma 39. *Given X a successful trailing instance of* take *or* steal*:*

$$r^{\mathrm{t}} = t(m) \in X \implies m \geq m_{\mathrm{F}}$$

Proof. We have two cases:

- Assume X is an instance of *take*. X is sequenced after P_{F}: $r^{\mathrm{t}}_{P_{\mathrm{F}}} = t(m_{\mathrm{F}}) \xrightarrow{\mathrm{sb}} r^{\mathrm{t}}_X = t(m)$, and $m \geq m_{\mathrm{F}}$ by sequential constraints.
- Assume X is an instance of *steal*. Since X is successful, X contains a successful instance of compare-and-swap $s^{\mathrm{t}}_X = t(m) - 1 \rightarrow t(m)$. If $m < m_{\mathrm{F}}$, then:

$$\mathrm{sc}\, s^{\mathrm{t}}_X \xrightarrow{\mathrm{rf}} \cdots \xrightarrow{\mathrm{rf}} \mathrm{sc}\, s^{\mathrm{t}} = t(m_{\mathrm{F}}) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{t}}_{P_{\mathrm{F}}} \xrightarrow{\mathrm{sb}} w^{\mathrm{b}} = b(n) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_X$$

  With $w^{\mathrm{b}} = b(n)$ either a release itself or in a release sequence that begins after $r^{\mathrm{t}}_{P_{\mathrm{F}}}$. This is a cycle in happens-before, which is not allowed. Therefore, $m \geq m_{\mathrm{F}}$. □

Lemma 40. *Given X and Y distinct successful trailing instances of* take *or* steal*, then:*

$$\forall r = x_i(v) \in X,\ \forall r' = x_{i'}(v') \in Y,\ i \neq i'$$

Proof. Assume $i = i'$. By Theorem 6, we have $v \neq v'$. Thus there exist two distinct significant writes $w = v_i(v)$ and $w' = v_{i'=i}(v')$, which provide the two versions.

Without loss of generality, let us assume $v < v'$. By definition, $w^{\mathrm{b}}_{P_{\mathrm{F}}} = b(n_{\mathrm{F}})$ is sequenced after both writes. Furthermore, there is a release fence in *give* after each significant write, thus before $w^{b(n_{\mathrm{F}})}_{P_{\mathrm{F}}}$. Since X reads from $w^{\mathrm{b}}_{P_{\mathrm{F}}}$, we have:

$$w' = x_{i'=i}(v') \xrightarrow{\mathrm{sb}} \mathrm{rel}\, w^{\mathrm{b}}_{P_{\mathrm{F}}} = b(n_{\mathrm{F}})\ (\xrightarrow{\mathrm{sb}} w^{\mathrm{b}}) \xrightarrow{\mathrm{rf}} \mathrm{acq}\, r^{\mathrm{b}}_X \xrightarrow{\mathrm{sb}} r^{\mathrm{a}} \xrightarrow{\mathrm{sb}} r = x_i(v)$$

With w^{b} either a release itself or in a release sequence that begins with or after $w^{\mathrm{b}}_{P_{\mathrm{F}}}$. Hence we have $w' \xrightarrow{\mathrm{hb}} r^{\mathrm{a}} \xrightarrow{\mathrm{sb}} r$, and it follows from Lemma 33 that $v' \leq v$; thus, $v < v' \leq v$. Impossible. □

Corollary 12. *The combined number of successful trailing instances of* take *and* steal *is less than or equal to $b(n_{\mathrm{F}}) - t(m_{\mathrm{F}})$.*

Proof. Let X be a successful trailing instance of *take* or *steal*, with $r^{\mathrm{b}}_X = b(n)$ and $r^{\mathrm{t}}_X = t(m)$. We know that $n \geq n_{\mathrm{F}}$ (by definition) and $m \geq m_{\mathrm{F}}$ (from Lemma 39).

Furthermore, a *take* operation always contains one decrementing write to *bottom* (by one), which may be followed by one incrementing write to *bottom* (by one). Hence $n \geq n_{\mathrm{F}}$ implies $b(n) \leq b(n_{\mathrm{F}})$.

Therefore, X can only read at an index i, such that $t(m_{\mathrm{F}}) \leq i < b(n_{\mathrm{F}})$. Lemma 40 tells there can be no more than $b(n_{\mathrm{F}}) - t(m_{\mathrm{F}})$ such X. □

Lemma 41. *There is a finite number of successful (trailing or non-trailing) instances of* take *or* steal.

Proof. It follows from Corollary 12 that there is a finite number of successful trailing instances of *take* or *steal.*

Furthermore, there must be a finite number of non-trailing *take* operations, which are sequenced before P_{F}.

Lastly, there is a finite number of *give* operations, thus *bottom* reaches a maximum $b_{\max}$ over the execution of the program. Since two successful *steal* operations must read different values of *top*, there can be no more than $b_{\max}$ successful instances of *steal.*

Hence the finite number of successful instances of *take* or *steal.* □

Lemma 42. *In each thread, there exists X a failed instance of* take *or* steal *such that:*

$$\forall r^{\mathrm{b}} = b(n) \in X,\ \forall r^{\mathrm{t}} = t(m) \in X,\ b(n) \leq t(m)$$

Furthermore, each thread makes no more than $1 + m_{\mathrm{F}} + b(n_{\mathrm{F}}) - t(m_{\mathrm{F}})$ attempts at take *or* steal *that result in a failed compare-and-swap instruction.*[1]

Proof. It follows from Lemma 41 that there is a finite number of successful instances, hence a finite number per thread. Thus, there must exist a failed instance of *take* or *steal.*

A failure can occur either because the deque is empty ($b(n) \leq t(m)$) or because of a failed compare-and-swap instruction. Suppose there is no X where $b(n) \leq t(m)$; then all failures must be due to a failed compare-and-swap instruction. A failed compare-and-swap occurs if the two values of *top* read during the instance X differ. Let Y_1 and Y_2 be two such failed instances executing in a same thread; let us assume that Y_2 is sequenced after Y_1. For some $m_1 \neq m_1'$ and $m_2 \neq m_2'$:

$$r^{\mathrm{t}}_{\mathrm{Y}_1} = t(m_1) \xrightarrow{\mathrm{sb}} r^{\mathrm{t}} = t(m_1') \xrightarrow{\mathrm{sb}} r^{\mathrm{t}}_{\mathrm{Y}_2} = t(m_2) \xrightarrow{\mathrm{sb}} r^{\mathrm{t}} = t(m_2')$$

There exists a write $s^{\mathrm{t}} = t(m_1') \xrightarrow{\mathrm{rf}} r^{\mathrm{t}} = t(m_1') \xrightarrow{\mathrm{sb}} r^{\mathrm{t}}_{\mathrm{Y}_2} = t(m_2)$. Therefore, $m_1' \leq m_2$.

Since $m_1 \neq m_1' \ \wedge \ m_2 \neq m_2'$, and *top* is monotonically increasing, it must be that $t(m_1) < t(m_1') \leq t(m_2) < t(m_2')$. It follows from Corollary 12 that *top* takes no more than $1 + m_{\mathrm{F}} + b(n_{\mathrm{F}}) - t(m_{\mathrm{F}})$ different values.

Therefore, there can be no more than $1+m_{\mathrm{F}}+b(n_{\mathrm{F}})-t(m_{\mathrm{F}})$ compare-and-swap-failing instances of *take* or *steal* per thread. Since there is also a finite number of successful such instances, any further *take* or *steal* operations must return empty, and the thread reaches its stationary point. □

Corollary 13. *The combined number of successful (trailing or not) instances of* take *and* steal *is equal to the number of* give.

Proof. A successful instance of *take* either decreases the value of *bottom* by one or increases the value of *top* by one; a successful instance of *steal* increases the value of *top* by one. An instance of *give* increases the value of *bottom* by one.

[1] Hence a thread eventually reaches a stationary state where $b = t$; it should be noted that the model does not guarantee progress; it is legal for a thread to end up looping on a non-final state where $b = t$ but $b \neq b(n_{\mathrm{F}})$.

It follows from the previous lemma that the worker thread reaches a stationary point where $b = t$. Clearly, this cannot occur before all *give* operations and all successful instances of *take* have occurred.

Since $b = t$ at the stationary point and all increases to *bottom* precede, the sum of increases to *top* and decreases to *bottom* (the combined number of successful instances of *take* and *steal*) must be at least equal to the number of increases to *bottom* (the number of *give* operations).

It is exactly equal, as otherwise there would be more significant reads than significant writes, which is impossible according to Theorem 6. □

One may finally prove Theorem 7. On the one hand, Corollary 13 tells that the number of significant reads (from a successful instance of *take* or *steal*) is equal to the number of significant writes (from an instance of *give*). On the other hand, Theorem 6 states that significant reads uniquely map to significant writes. By injectivity, there exists a unique significant read for each significant write.

Appendix B

Reading the libkpn source code

This annex serves as a brief reading guide to the C11 source code of the libkpn high-performance blocking run-time library, whose package should have been distributed alongside this thesis document. It is a continuation of Section 4.3.2, which explains the high-level design and correspondence with the algorithms described in earlier sections.

This implementation has been tested extensively[1] on Linux and MacOS X, on x86-64 and ARMv7, and a variety of compilers including several versions of the GNU, Intel, and Clang compilers.

The package should contain the following main C files:

log.c, logtr.c, log.h A low-overhead concurrent logging system, for debugging and profiling purposes. See Appendix B.2.

kpn.h Main header file, to be included by client programs.

sched.c, sched.h Main process scheduling code for the primary scheduler, with passive waiting, as described in Section 4.1.3.1. Also includes a custom (manual) garbage collection and termination-detection algorithm based on reference counters, not described in this thesis, and which supports incremental reconfiguration, as suggested in Section 4.3.1.

spscq.c, spscq.h Caching single-producer single-consumer queues, which implement the algorithm described in Section 4.2.2.

synctask.c, synctask.h Interface between Kahn processes and external (non-worker) POSIX threads, mutexes and signals.

tarray.c, tdeque.c, tarray.h, tdeque.h Chase–Lev work-stealing deque implementation, which closely follows Section 4.2.3.

task.c, task.h Bookkeeping functions for Kahn process objects.

thrpool.c, thrpool.h Simple thread pools, to represent workers.

[1] Although, as stated by the license, it is provided without any express or implied warranties.

types.h All public types defined by the library.

The following source and header files define utility functions that are referenced by the main sources:

a.h Convenience macros for atomic operations.

atomic.h, stdatomic.h Drop-in replacement of *stdatomic.h* for select older compilers, and auto-selection header based on compile-time constants.

c11.h, thread_local.h Stubs for non-essential C11 features that might not be present in all compilers.

cachealign.c, cachealign.h Cache alignment and memory allocation.

cotask.h, ... (Multiple files beginning with the *cotask* prefix.) Alternative encoding for step functions, as single interruptible stateful coroutines. Those support context switches using low-level assembly or the *ucontext.h* Unix library. None of our examples actually use this.

nanotime.h Access to a precise wall-clock, for tests and benchmarks.

B.1 Libkpn architecture overview

The libkpn run-time library is built around Kahn processes (*KPN_TASK* objects), internally referred to as tasks, for historical reasons, channels (*KPN_SPSCQ* objects), and scheduling data associated with each worker thread (*KPN_WORKER*).

Each process holds the argument channels it is bound to in arrays, in the *_qvs* field of the *KPN_TASK* structure. Just like the non-blocking interpreter designed in Chapter 3, libkpn channels do not normally link directly to the processes to which they are bound. Consequently, it is not possible to query a channel to know who is writing or reading on the other side.

Processes that are ready to schedule (which make up the W set, according to Section 4.1.2) are distributed among work-stealing deques, of type *KPN_TDEQUE*.

Pointers to active processes (the A set), currently executing on some worker, are removed from shared memory completely, giving exclusive rights to said worker.

Lastly, sleeping processes (the D set) are stored in *_sa* field of the opposite process, on which awakening is dependent. Positions in *_sa* and *_qvs* match. Channel objects are attached to processes by having their *_s* field set to refer to the appropriate slot in a *_sa* array, making it possible for another process to wait on the opposite side of the queue.

The run-time library supports two passive waiting schemes, as well as several variants, which can be chosen at run-time by calling the *kpn_set_stall_policy* function. The first, *KPN_STALL_FENCE*, implements a double check after each successful push or pop, which requires a more expensive sequentially consistent fence for every operation.

The second, *KPN_STALL_SCAN*, uses a lazy waiting strategy, which relies on a full scan of adjacent nodes before putting a process to sleep. Both methods are described in Section 4.1.3.1. We thus forgo the fence after push and pop operation, however, as we have seen, there is a risk of being overzealous and waking up unnecessary processes while walking through neighbors.

The representation of the D set as arrays in the responsible process is a deliberate design choice that significantly speeds up the lazy approach, which makes it a worthwhile choice for some workloads. Others benefit more from the regularity of the systematic-fence strategy.

B.2 Logging and tooling

Libkpn comes with a logging system designed to have minimal interference with memory orderings and system thread scheduling. We use thread-local buffers, periodically flushed to disk on separate files, to be joined later by an external script. The logging format is binary, which eliminates the need to serialize data; an external program, *logtr*, is used to parse these binary dumps and convert them to SQL entries,[2] for statistics, analyses and visualization.

The implementation makes extensive use of this logging framework, which also acts as the main debugging tool available to diagnose and repair problems in the library itself. More elaborate traditional tools such as debuggers (e.g., GDB) and memory tracers (e.g., Valgrind) tend to serialize thread actions, due to the necessary memory and system hooks used. As such, most issues related to race conditions and the memory model have been caught and solved using our custom tool set rather than more standard means.

The *tools* folder at the base of package contains a collection of scripts to be used in conjunction with log files. The main utility is known as *kpn_prof* and serves as a driver for several other tools. It has subcommands to enable logging in libkpn-enabled programs, printing the resulting logs, extracting logs from a core dump, and exporting to SQL.

Once extracted to SQL, the *timeline.tcl* visualization tool displays an interactive time line of process activation, as color-coded tasks, on the different worker threads. The user can then browse scheduling events, and zoom around to get a feel of the actual concrete execution, as well as possible bottlenecks.

[2] Those are stored in SQLite database files, which can be manipulated without a server.

Résumé

La thèse porte sur les réseaux de Kahn, un modèle de concurrence simple et expressif proposé par Gilles Kahn dans les années 70, et leur implémentation sur des architectures multi-cœurs modernes, à mémoire partagée. Dans un réseau de Kahn, le programmeur décrit un programme parallèle comme un ensemble de processus et de canaux communicants, reliant chacun exactement un processus producteur à un consommateur.

Nous nous concentrons ici sur les aspects algorithmiques et les choix de conception liés à l'implémentation, avec deux points clefs : les garanties non bloquantes et la mémoire relâchée. Le développement d'algorithmes non bloquants efficaces s'inscrit dans une optique de gestion des ressources et de garantie de performance sur les plateformes à ordonnancement irrégulier, telles que les machines virtuelles ou les GPU. Un travail complémentaire sur les modèles de mémoire relâchée vient compléter cette approche théorique par un prolongement plus pratique dans le monde des architectures à mémoire partagée contemporaines.

Nous présentons un nouvel algorithme non bloquant pour l'interprétation de réseaux de Kahn. Celui-ci est parallèle sur les accès disjoints : il permet à plusieurs processeurs de travailler simultanément sur un même réseau de Kahn partagé, tout en exploitant le parallélisme entre processus indépendants. Il offre dans le même temps des garanties de progrès non bloquant : en mémoire bornée et en présence de retards sur les processeurs. L'ensemble forme, à notre connaissance, le premier système complètement non bloquant de cette envergure : techniques classiques de programmation non bloquante et contributions spécifiques aux réseaux de Kahn. Nous discutons également d'une variante bloquante destinée au calcul haute performance, avec des résultats expérimentaux encourageants.

Mots Clés

Concurrence, parallélisme, programmation non bloquante, modèles de mémoire relâchée, réseaux de processus de Kahn

Abstract

In this thesis, we are interested in Kahn process networks, a simple yet expressive model of concurrency, and its parallel implementation on modern shared-memory architectures. Kahn process networks expose concurrency to the programmer through an arrangement of sequential processes and single-producer single-consumer channels

The focus is on the implementation aspects. Of particular importance to our study are two parameters: lock freedom and relaxed memory. The development of fast and efficient lock-free algorithms ties into concerns of controlled resource consumption and reliable performance on current and future platforms with unfair or skewed scheduling such as virtual machines and GPUs. Our work with relaxed memory models complements this more theoretical approach by offering a window into realistic shared-memory architectures.

We present a new lock-free algorithm for a Kahn process network interpreter. It is disjoint-access parallel: we allow multiple threads to work on the same shared Kahn process network, fully utilizing the parallelism exhibited by independent processes. It is non-blocking in that it guarantees global progress in bounded memory, even in the presence of (possibly infinite) delays affecting the executing threads. To our knowledge, it is the first lock-free system of this size, and integrates various well-known non-blocking techniques and concepts (e.g., safe memory reclamation, multi-word updates, assistance) with ideas and optimizations specific to the Kahn network setting. We also discuss a variant of the algorithm, which is blocking and targeted at high-performance computing, with encouraging experimental results.

Keywords

Concurrency, parallelism, lock-free programming, relaxed memory models, Kahn process networks