Crem user's manuel

Jocelyne Erhel

INRIA

CENTRE DE ROCQUENCOURT

Rapports Techniques

N° 25

CREM USER'S MANUAL

Jocelyne ERHEL

Mai 1983

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

CREM USER'S MANUAL

Jocelyne ERHEL

I.N.R.IA.
Domaine de Voluceau
Rocquencourt,
B.P. 105
78153 LE CHESNAY CEDEX

France

RESUME : Ce manuel decrit le logiciel CREM developpe a l'INRIA sur l'ordinateur CII-HB/DPS68 dote du systeme MULTICS. Ce programme emule les primitives multitaches proposees pour CFT (Cray-XMP,...). On peut ainsi utiliser un environnement multi-process sur MULTICS. Le manuel donne les regles d'utilisation de CREM, ainsi que quelques exemples.

ABSTRACT : This manual describes the Software CREM developped at INRIA on CII-HB/DPS68 with MULTICS System to emulate the CFT Multi-tasking facility. This allows to utilize multi-process environment on MULTICS System. Rules and examples of use are provided.

CONTENTS

SECTION 1

SYSTEM ENVIRONMENT

1) Using the Multi-tasking facility

The Multics System provides multiprocess environment. Concurrent Processes may run on three CPU's, and can cooperate.

The system allows the user to create several Processes, within limits defined by System-Administrator. CREM enables the user to utilize the facility through FORTRAN codes. It is transparent to the FORTRAN Language.

The user's code is partitioned into a set of TASKS. Each task is allocated into a Process. The programmer must create himself these partitions and control their interaction. Basic functions are provided by the Software, which emulate the proposed multi-tasking facility for CFT.

2) MultiProcess system

The user must create the Processes by interactive or absentee commands, which are explained in details in Section 4. The number of Processes is fixed throughout all the execution of the job.

The Processes communicate through a segment called the SEGCOM (Segment of Communication) which is initialized in a first step(cf. section 4). Its structure is described in Annexe 1. In particular SEGCOM contains the parameters of the system. The user accesses SEGCOM only through the Software in a transparent fashion.

The Software uses Multics System facilities to synchronize the Processes. In particular, it utilizes :

- Inter Process Communication

- Locking Mechanism (only the builtin function STAC but not the routine SET_LOCK).

See Section 5 for more details.

The Software allocates each user task into a Process, and does not wait for a Process to become free. A Process can be reused after RETURN of a task. The number of concurrent tasks cannot therefore exceed the number of participating Processes.

All Processes are equal in Multics System. But from the user's point of view, the main program is a Mother Task which creates the first subtask. Therefore the Software introduces the notion of Master/Slaves Processes.

- The master Process executes the main program
- The slaves Processes execute other tasks.

The user's job begins execution only when all participating Processes are logged in. The main program is allocated into the master Process, while other slave Processes are waiting for a task. The job stops when the main program returns. Slave Processes are logged in until reception of a STOP message sent by the Master Process, at the end of the main program.

SECTION 2

FORTRAN ENVIRONMENT

1) Subroutines and tasks

Task control functions enable the programmer to start tasks and wait for their completion (see Section 3).

All tasks -main included- are standard FORTRAN subrcutines, which are called by the Software, after allocation into a free Process.

After a RETURN statement of a subtask, the Software frees the allocated slave Process and wakes up the Processes waiting for completion of the task.

After a RETURN statement of the main task, the Software controls that all slave Processes are free. In that case, it stops all participating Processes.

Inside a task, subroutines are called by the FORTRAN CALL statement and are executed on the same Process.

2) Shared variables

The concurrent tasks communicate through shared variables. In Multics, each Process has :

- its own address space
- its own stack(s) of variables
- access to any segment, provided it is made known and access control conditions are satisfied.

Hence, the Processes can share variables only through externally known segments, accessed in FORTRAN by use of a labelled COMMON whose name begins with a $.(COMMON/name$/...) All shared variables are therefore declared in a "$" COMMON.

In particular, the arguments of a task transmitted by address must be stored in shared segments. Otherwise different Processes could not access them.

The pathnames of the shared segments are contained in a table which facilitates and speeds up address transmission from one Process to another (see Section 5). This table is initialized with the Segment of Communication. The user must

declare all shared segments where parameters of tasks are stored. (It is not necessary to declare segments referenced in FORTRAN only through "$" COMMON).

3) Compiler considerations

It is necessary to ensure the proper updating and transmission of shared variables between cooperating tasks. The following policy :

- makes no assumption on compiler optimizations, including code movement or removal;
- allows some flexibility in parameter transmission within a task.

Any variation not in accordance to the following rules is NOT allowed in this code. The simple fact that another coding policy works in one implementation with a particular compiler will not be considered as a proof of this program's validity.

a) Sending shared variables to other Processes :

The only two occasions in a program where it can be assumed that a shared variable has been updated in storage and is accessible from other tasks are :

(i) - The variable is declared in a "..$" COMMON by the updating program unit.

- A CALL to an external procedure has occured, after the local updating of the variable in that program unit. Language defined functions that are known to the compiler cannot be regarded as external procedure in this case.

(ii) - The variable satisfies the above requirements in program unit <A>.The variable's updating occurs in program unit <B> called, possibly indirectly, by <A>. Parameter passing from <A> to <B> can be made by any means local to a task, within FORTRAN rules.

In such a case, the shared variable will be assumed updated in shared storage after :

- program unit <B> has returned
- program unit <A> has performed a call to an external procedure, in accordance to rule (i) after <B>'s RETURN.

b) Receiving shared variables from other Processes :

To access a shared variable, that may have been updated by another Process, any program unit first has to receive a signal that the variable updating has been performed. At this point, the accessing program unit has to obtain a fresh copy of the data. This must be done by :

- Having the shared variable declared in the "..$" COMMON
- Performing a call to an external program unit after the reception of the synchronisation signal and before accessing the data. Calling an external program unit that is known to return upon reception of this signal is sufficient: this will be one of the synchronization routines. See Section 4 for examples of synchronizations.

Although they are known to the CRAY compiler, the following procedures can be regarded as external program units:

TSKSTART,TSKTEST,TSKWAIT
LOCK,LOCKASGN,LOCKREL,TLOCK,UNLOCK
EVASGN,EVCLEAR,EVPOST,EVREL,EVTEST,EVWAIT

4) I/O library

Each Multics Process owns its user_input (absin) and user_output (absout). They can also share files, provided Acces_Control conditions are satisfied. The Multics System sets adequate interlocks. Only necessary synchronizations occur while performing simultaneous I/O operations on conflicting files.

The attach description of a shared file has the following form :

vfile_ path -append -blocked {N} -share {N}
where : path is the absolute or relative pathname -append in input_outout openings, this control argument causes put_chars and write_records operations to add to end of file instead of truncating when the file position is not

at end of file. Also the position is initially set to beginning of file, and an existing file is not truncated at open.

-share {N}
allows a file to be open in more than one process at the same time, even though not all openings are for input. If specified, N is the maximum time that this process waits to perform an operation on the file. A value of -1 means the process may wait indefinitely. the default value of N is 1.

-blocked {N}
specifies attachment to a blocked file. If a nonempty file exists, N is ignored and may be omitted. Otherwise, N is used to set the maximum record size (bytes).

SECTION 3

THE MULTI-TASKING LIBRARY

The Software emulates the library provided by the "proposed multi-tasking facility for CFT".

To provide the minimal set of functions required for user directed multi-tasking, three separate types of facilities will be provided. They are :

- Task control
- Lock control
- Event control

Some slight differences appear between the CRAY library and their emulations, due do Multics system considerations and design rules. (See Section 5 and Notes on the CRAY-XMP system). They are noted with %%.

1) TASK control

Three library routines will be provided : a subroutine to initiate a task ; a subroutine to wait for completion of a task ; a function to determine whether a task already exists.

a) To initiate a task the programmer needs the routine TSKSTART.

%% The maximal number of arguments of a task is a parameter of the Software.

%% The arguments can be transmitted :

- by adress : in that case, they must be declared in a "$" COMMON and stored in shared segments, so that any Process can adress and access them.
- by value : for sake of simplicity, one word variables (integer, real simple precision, logical) can be passed by value. The user may therefore transmit variables stored locally in a Process to another Process.

%% Two syntaxes are used :

```
CALL TSKSTART (<task_control_array>,
               <subroutine_name>,
               <by_adress_argument_list>)
CALL TSKSTART (<task_control_array>,
               <number_of_by_value_arguments>,
               <subroutine_name>,
               <by_value_argument_list>,
               <by_adress_argument_list>)
```

The task_control_array must be unique for each active task the user creates.
This array can contain control information that is used by the multitasking library to control the execution of the specific task. two entries are currently reserved in this array: the first word of the array must contain the total number of words in the array - at least 2; and the second word is filled in by the multitasking library with the unique task identifier. Aditionnal entries may be defined in the array as needs are identified. However, upward compatibility will be maintained because of the ability to determine the number of entries the user has provided. The programmer will need to use this identifier when waiting on task completion or determining whether a task is still executing.

The <subroutine-name> is the external entry point that contains the code for the task. Because of the design of the FORTRAN language, the programmer will also need an EXTERNAL statement in the program for this entry point.

%% The <by-adress - argument - list> and the <by-value - argument - list> are the list of parameters that needs to be passed by adress or by value to the new task when the <subroutine-name> is entered. What, if anything, is passed must be determined by the programmer.

%% The <number - of - by - value - arguments> must be equal to the effective length of <by - value - argument - list>. Otherwise, errors can occur that cannot be detected by the Software.

EXAMPLES :

```
CALL TSKSTART (ITSK1,CGPAR)
CALL TSKSTART (ITSK,CGPAR,NDIM,AMAT,VECT)
CALL TSKSTART (ITSK,2,ASMEF,NTRI,NPT,A)
```

b) To wait for completion of a task the programmer needs the following statement :

```
CALL TSKWAIT (<task_control_array>)
```

The task_control_array must contain the same information it had on return from the initial TSKSTART call.
The execution of the program will be suspended until the namec task completes execution.

```
EXAMPLE :
CALL TSKWAIT (ITSK)
```

c) To determine whether a task exists the programmer can use the function :

```
TSKTEST (<task_control_array>)
```

The task_control_array must contain the same information it had on return from the initial TSKSTART call.
This function will return a logical value, so the programmer must also include a LOGICAL type declaration for it in the program.
A logical "TRUE" is returned if a task exists with an identifier that matches the task identifier in the task_control_array. This result will be returned no matter what state of execution the task is in.
A logical "FALSE" is returned if the task was never created or was created and has completed execution.

```
EXAMPLE :
LOGICAL TSKTEST
IF (TSKTEST(ITSK)) GOTO 10
```

2) LOCK CONTROL

Five library routines will be provided : a subroutine to assign a unique lock identifier ; a subroutine to set a lock ; a subroutine to clear a lock ; a function to determine if a lock is set ; and a subroutine to release a lock identifier.

a) To cause a unique lock identifier to be created the programmer needs the following statement :

```
CALL LOCKASGN (<integer-variable>)
```

The multi-tasking support library will create an unused identifier that will be returned in the <integer-variable> and can be used by the program as a lock identifier to set, clear, or test a lock.

%% The Software manages the locks by means of a table of limited size. The maximal number of assigned locks is therefore a parameter.

EXAMPLE :
CALL LOCKASGN (LOCKID)

b) To set a lock the programmer needs the following statement :

CALL LOCKON (<integer_variable>)

The integer_variable will be set with a unique value that indicates that the lock is in the locked stat. This variable must have been initialized by a previous call to LOCKASGN.

%% The integer_variable is positive when it is initialized or cleared, and is negative when it is set.

The function of this routine is to set a lock and return execution to the calling program. If the lock has already been set, the calling program is suspended until the lock has been cleared by another task and can be set by this one.

%% The waiting tasks are partially ordered in a queue in order to avoid any starvation problem.

EXAMPLE :
CALL LOCKON (LOCKID)

c) To clear a lock the programmer needs the following statement :

CALL LOCKOFF (<integer>)

The integer_variable will be set with a unique value that indicates that the lock is in the unlocked stat. This variable must have been initialized by a previous call to LOCKASGN.

%% The integer_variable is positive when it is initialized or cleared, and is negative when it is set.

The function of this routine is to clear a lock so that other tasks may have access to it.

EXAMPLE :
CALL LOCKOFF (LOCKID)

d) To determine if a lock has already been set the programmer needs the following function :

LOCKTEST (<integer>)

The value in the integer_variable is tested to determine if it is in the locked state. If the variable is in the unlocked state, it is changed to the locked state. This variable must have been initialized by a previous call to LOCKASGN.

If the integer_variable was originally in the locked state, this function will return a logical "TRUE" value. If not, it is placed in the locked state and a logical "FALSE" value is returned.

Since the data type for this function does not match the normal FORTRAN default, the programmer will need a LOGICAL LOCKTEST type declaration in the program.

EXAMPLE :
LOGICAL LOCKTEST
IF (LOCKTEST (LOCKID)) GOTO 10

e) To release a unique identifier that has been created by a LOCKASGN call the programmer needs the following statement :

CALL LOCKREL (<integer_variable>)

The value of the integer_variable must be the same value returned from the LOCKASGN call.

The function of this routine is to detect errors that may arise when a task is waiting on a lock that will never again be cleared. Any reference to this identifier while it

remains unassigned is an error, although it may again be used after another call to LOCKASGN.

EXAMPLE :
CALL LOCKREL (LOCKID)

3) EVENT control

Six library routines will be provided to manage EVENTS : a subroutine to define an event ; a subroutine to wait for an event to be posted ; a subroutine to post an event ; a subroutine to clear an event ; a function to determine if an event has been posted ; and a subroutine to remove the definition of an event.

a) To allocate a unique identifier and cause an event to become defined, the programmer needs the following statement :

CALL EVASGN (<integer-variable>)

The multi-tasking support library will create an unused identifier that will be returned in the <integer-variable> and can be used by the program as an event identifier to post, clear, wait on, or test an event.

%% The Software manages the events by means of a table of limited size. The maximal number of created events is therefore a parameter.

EXAMPLE :
CALL EVASGN (IEV)

b) To wait for an event to be posted by another task the programmer needs the following statement :

CALL EVWAIT (<integer-variable>)

The contents of the <integer-variable> must be the value generated by the multi-tasking support library on a call to EVASGN.
The function of this routine is to suspend execution of the task until the named event has been posted.

Any number of tasks may wait for the same event to be posted.

EXAMPLE :
CALL EVWAIT (IEV)

c) To post an event the programmer needs the following statement :

CALL EVPOST (<integer-variable>)

The contents of the <integer-variable> must be the value generated by the multi-tasking support library on a call to EVASGN.
The function of this routine is to mark an event as posted and to cause all tasks waiting on that event to resume execution.

It is allowed to post an event already posted.

EXAMPLE :
CALL EVPOST (IEV)

d) To clear an event the user needs the following statement :

CALL EVCLEAR (<integer-variable>)

The contents of the <integer-variable> must be the value generated by the multi-tasking support library on a call to EVASGN.
The function of this routine is to remove an event from the posted state.

It is allowed to clear an event not yet posted.

EXAMPLE :
CALL EVCLEAR (IEV)

e) To determine whether an event has been posted the programmer needs the following function :

EVTEST (<integer-variable>)

The contents of the <integer-variable> must be the value generated by the multi-tasking support library on a call to EVASGN.

%% Otherwise it is an error.

This is a logical function and the programmer will also need a LOGICAL EVTEST type declaration for it in the program.

This function will return the value "TRUE" if the event has been posted and the value "FALSE" if the event has never been posted or has been cleared.

```
EXAMPLE :
LOGICAL EVTEST
IF (EVTEST (IEV)) GOTO 10
```

f) To release an event identifier that was defined with an EVASGN call, the programmer needs the following statement :

```
CALL EVREL (<integer-variable>)
```

The contents of the <integer-variable> must be the value generated by the multi-tasking support library on a call to EVASGN.

The function of this routine is to return the identifier to the pool of undefined events for re-allocation at some future call to EVASGN. Any reference to this identifier while it remains unassigned is an error.

```
EXAMPLE :
CALL EVREL (IEV)
```

SECTION 4 :

UTILIZATION OF CREM

1) FORTRAN coding

The user writes standard FORTRAN codes, with the following restrictions :

- The main program must be a SUBROUTINE without parameter. In fact, it is called by the Software.
- The library routines used must be declared, since they are PL/1 procedures :

```
EXTERNAL TSKSTART (DESCRIPTORS)
EXTERNAL LOCK     (DESCRIPTORS)
...
```

or

```
%include cftmtl
```

The include file cftmtl.incl.fortran (for CFT Multi Tasking Library) contains all those external declarations.

- In general, the main FORTRAN must contain sufficient calls to TSKWAIT to guarantee that all slaves Processes are free when the master Process stops execution.

2) Initialization

The segment of communication SEGCOM must be initialized in a preparation step. It is made by the procedure crem_init

Usage :

```
crem_init <SEGNAME>
```

where SEGNAME is the name of SEGCOM.
The segment is created or updated in the working-directory.

REMARK : The segment is created if it does not exist.

The procedure crem_init initializes the structure SEGCOM and defines the parameters of the Software. Default parameters are provided, which may be changed by the user. The shared segments (containing arguments of TSKSTART) must be declared. The user gives their number and their pathnames.

REMARK : A new initialization is not necessary, for reuse after a normal end of the job. SEGCOM is then automatically reinitialized. But it must be done after job abort. (The SEGCOM is not deleted after job abort in order to allow the user to dump it and analyze at will).

PARAMETER	DEFAULT VALUE	COMMENT
n_Process	2	number of logged in Processes
n_lock	10	maximum number of simultaneously assigned locks
n_events	10	maximum number of simultaneously assigned events
n_arg	10	maximum length of argument list in TSKSTART
n_path	0	number of shared segments.

LIST OF PARAMETERS

The use of COMMON with $ is standard :

USAGE :
cr NAME
sfc NAME
COMMON/NAME$/...

3) Creation of the Process

The Processes are logged in by interactive or absentee commands.

- Interactive commands : each connected user owns a Process. Different interactive user's Processes can cooperate, provided access control conditions are satisfied.
- Absentee commands : the Process is logged in the absentee queue C, so as to be connected with no delay. (All participating Processes must be logged in to begin execution). The command ear is used as usual.

EXAMPLES :

```
ear // -q 0 - li 300 -nt
ear test -q 0 - li 900 -tm 1:00
```

Of course, one Process can be interactive, while others are absentees.

4) Execution

The segment of communication must be initialized, and accessible by all participating Processes.
All Processes must have the same working_directory which must contain the SEGCOM.

The execution starts with the procedure crem_process. Two syntaxes enable the Software to distinguish the master Process from the slaves.

- crem_process <SEGNAME> <MAINNAME>

- crem_process <SEGNAME>

where SEGNAME is the name of SEGCOM and MAINNAME is the name of the main FORTRAN program.

REMARKS : - The main FORTRAN must be in the working_directory.
- One and only one Process has two arguments. An error occurs if not.

EXAMPLES :
```
crem_process SEGCOM
crem_process SEGCOM CGMAIN
```

5) Search rules

Software and Documentation are in the directory:

```
>udd>Menusin>Erhel>crem>v3
```

Synchronization library routines are in the directory:

```
>udd>Menusin>Erhel>crem>v3
>udd>Menusin>Erhel>crem>synch
```

They must therefore be in the user's search-list :

```
asr >udd>Menusin>Erhel>crem>v3
asr >udd>Menusin>Erhel>crem>synch
```

The SEGCOM and the MAIN FORTRAN must be in the home-directory. Search rules for other FORTRAN subroutines are standard.

The segment crem.info can help the user on line. It must be in the user's search-list info :

```
asc info >udd>Menusin>Erhel>crem>v3
```

USAGE : (standard)

```
help crem
```

6) Example of use

MASTER.ABSIN

```
asr >udd>Menusin>Erhel>crem>v3
cwd mult
hmu
crem_init SEGCOM
NON
.
ear slave -c 0 -nt
crem_process SEGCOM FTMAIN
```

SLAVE.ABSIN

```
asr >udd>Menusin>Erhel>crem>v3
cwd mult
hmu
crem_process SEGCOM
```

COMMAND :

```
ear master -q 0 -nt -tm "01:00"
```

Two Processes will be created like an interactive one at time 1:00 a.m. (in the limits of Multics load control group at this time!). The second Process is created by the first one, after initialization of the segcom.

7) I/O library

Tasks can open own files and shared files. The FORTRAN READ and WRITE instructions are the same. The files differ by their attach description.

attach description of a shared file:

```
io attach file(N) vfile_ path -append -share -blocked
```

where:
N is a switchname attached to the file and used in FORTRAN programs. path is the absolute or relative pathname of the file.

An output file must be modified after execution by editor, so as to get off all control characters.

EXAMPLE OF USE:

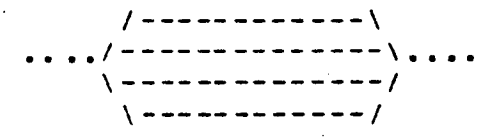
```
io attach file11 vfile_ synchro
     -append -share -blocked 100

subroutine test
...
write(11,100)
100     format(" this is a shared file")
...
```

8) Synchronization library

Some useful routines of synchronization are provided by the Software. They are :

a) FORKJOIN

It realizes the FORK AND JOIN of any number of Processes.



USAGE :

```
%INCLUDE 'MLBX'
CALL MLBXINIT
CALL FORKJOIN
```

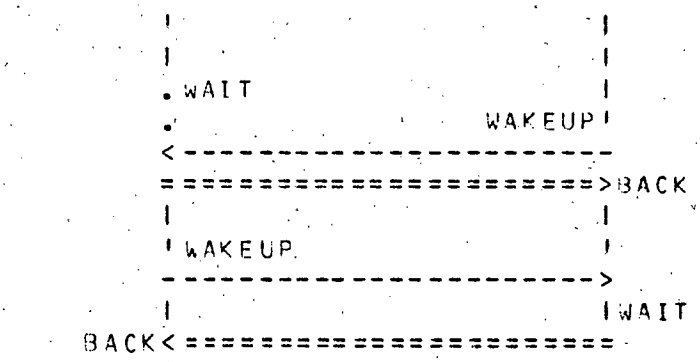
The sharec segment mlbx contains locks and events identifiers. It must be declared at the initialization step. The subroutine MLBXINIT must be executed one time before any call to FORKJOIN. It assigns locks and events identifiers. It also requires the number of tasks to be synchronized. (given by the user).

EXAMPLE :

```
      DO 1 I = 1,NITER
           CALL PARSOLVE
           CALL FORKJOIN
      1 CONTINUE
```

b) WAIT AND WAKEUP

They realize a RENDEZ-VOUS between two tasks:



WAIT blocks the task until reception of the WAKEUP. A WAKEUP cannot occur without a BACK of the previous one.

USAGE :

```
      DIMENSION MBRV(2,N)
      CALL RVINIT (N,MBRV)
      CALL WAIT (ME)                    CALL WAKEUP (OTHER)
```

MBRV contains the events identifiers. They are assigned by the routine rvinit. N is the number of tasks, 2 in general. ME is the number of the waiting task,while OTHER is the number of the waked task. We must have: 1 <= ME,OTHER <= N.

The user must take care of a proper correspondance between a WAIT and a WAKEUP to avoid deadlocks.

EXAMPLE :

```
          TASK 1                      TASK 2

          DO 1 I = 1,10               DO 1 I = 1,10
             CALL WAIT                   Y(I) = Y(I) + 1
             X(I) = X(I)+Y(I)            CALL WAKEUP
    1 CONTINUE                    1 CONTINUE
```

c) PRODCONS

This is an example of a system of producers/consumers. Buffers realize the interface between them.

USAGE :

PCSTART is the main program

PRODCONS is the task : CALL PRODCONS (INDEX)

The parameter INDEX distinghishes a producer from a consumer This program may be adapted to the actual needs of the user.

9) Debug

Some errors are signaled by the Software.The next version of CREM will contain some useful tools for debugging users' programs.

SECTION 5

NOTES OF IMPLEMENTATION

1) Design rules

The design of the Software was guided by several rules :

- to emulate as closely as possible the CFT multi-tasking facility
- to be easy to use
- to forbid active wait loops. The waiting Processes are systematically blocked.
- to avoid long critical sections. The locking mechanism is distributed among the variables of SEGCOM.
- to minimize calls to system subroutines.

2) Structures, variables and procedures

Two structures define the system in each Process. Comments identify them by different symbols in the Software codes :

- SEGCOM, noted $ in comments, is shared by all the participating Processes. A protection mechanism guarantees its coherence (by using the builtin function STAC).
- MY_PROCESS, noted & in comments, is an external structure known locally in the Process. It is described in Annexe 1.

The procedures use also parameters, noted > in comments, and local variables noted = in comments.

The pointers declared in the structure $SEGCOM$ have no other sense than a comment. The real pointers are declared in the structure &MY-PROCESS& or are local variables.

To speed up the access to the variables of $SEGCOM$, pointers defined by integer-offsets address the beginning of each table in SEGCOM.

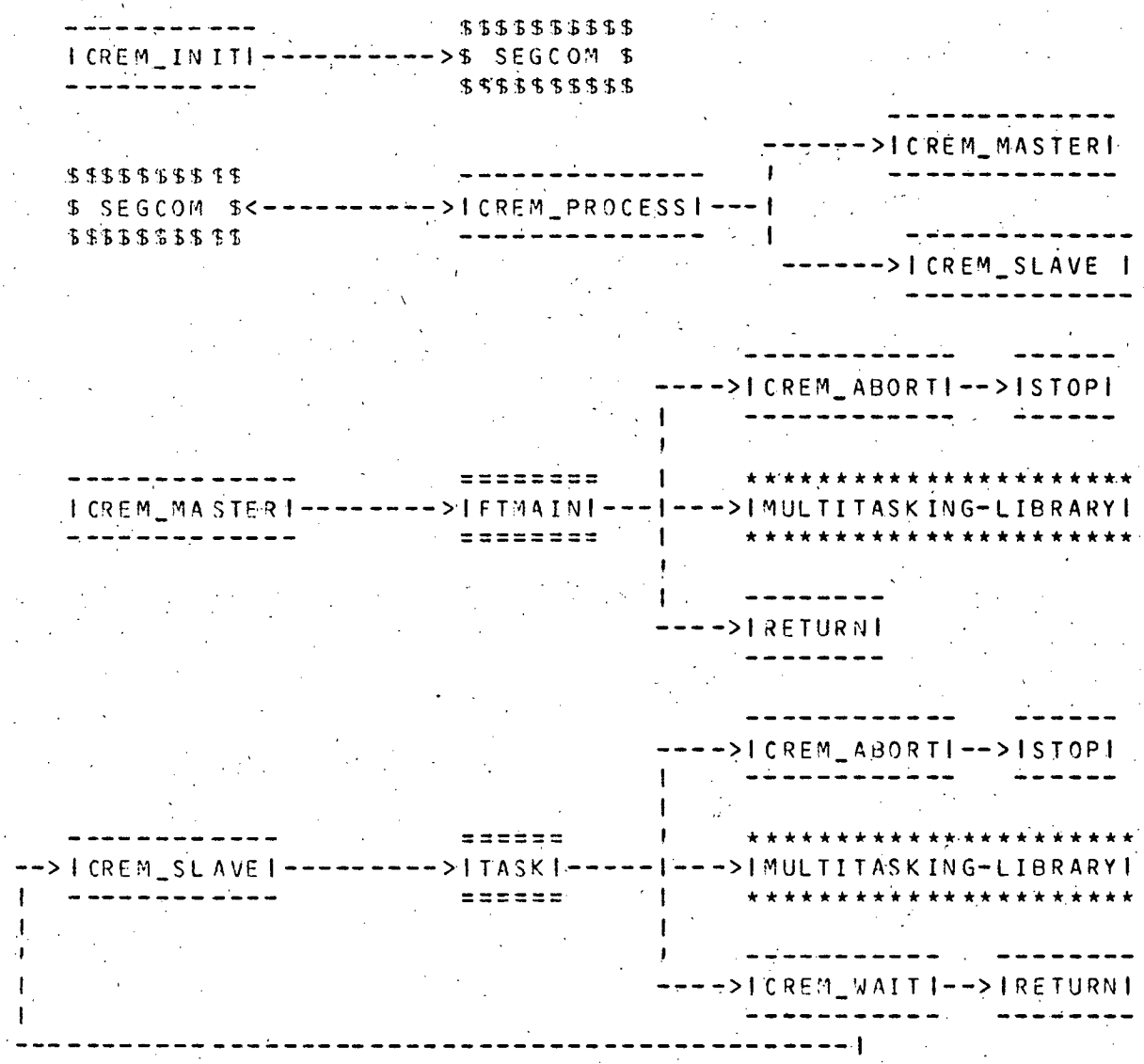
The Software provides the routines of the library for multi-tasking. They are written in PL/1 but can be

called by FORTRAN codes. Other procedures of the Software schecule the Processes. They are :

crem_init crem_process crem_master
crem_slave crem_wait crem_abort
pointer_receive pointer_send.

Comments and explanations are included in the PL/1 codes.

Organization :



3) Argument-list of a task

Two syntaxes are allowed to start a task on a new Process :

- TSKSTART (<task_id>, <subroutine_name>,
  <by_adress_argument_list>)
- TSKSTART (<task_id>,
  <number_of_by_value_arguments>,
  <subroutine_name>,
  <by_value_argument_list>,
  <by_adress_argument_list>)

The routine cu_$arg_list_ptr gives the pointer of the argument_list of TSKSTART :

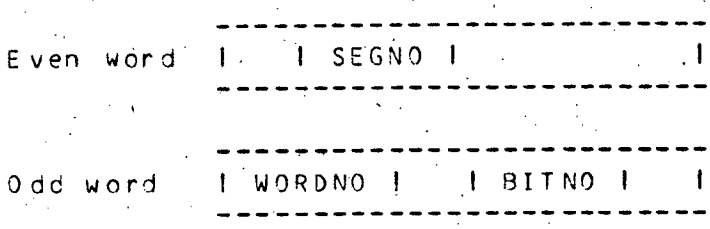
```
arg_list_ptr -> ------------------------------------------
                ! nb_arg    /* number of arguments */      !
                ! arg_ptrs  /* pointers of arguments */    !
                ! arg_desc  /* descriptors of arguments */ !
                ------------------------------------------
```

<subroutine_name> is identified by the entry descriptor. TSKSTART constructs the valid argument_list of the allocated Process. Pointers are local to Processes : See a). Therefore the Software must translate the entry and the arguments :

- <subroutine_name> is converted in a pathname : See b)
- values of by_value_arguments must be transmitted instead of their adress : See c).
- adresses of by_acress_arguments must be converted : See d)

a) Pointer and entry format

The pointer format is the following :



SEGNO is the segment number. WORDNO and BITNO are the offset of the pointer in the segment.

The segment number is assigned by the system to the Process and is local. The offset refers to the begin of the segment and is of course the same in any Process.

b) Entry translation

<subroutine_name> is an entry variable. But the entries are made known in each Process by different segment numbers. Therefore they must be translated before transmission to another Process. This is performed in two stages :

- The sending Process (which executes the TSKSTART procedure) translates the entry into a character string :
  * codeptr gives the pointer to the segment containing the entry.
  * hcs_$fs_get_path_name and get_entry_name give the pathname of the entry.
- The receiving Process (where the task is allocated) translates the character string into an entry :
  * cv_entry converts the pathname.

c) By value arguments

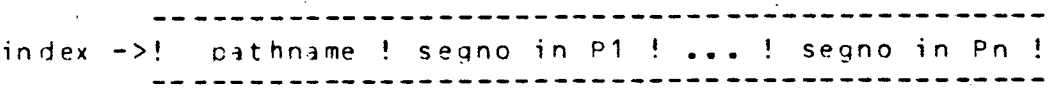
The transmitted argument_list must contain the value of by_value_arguments instead of their adress, which may be local to the Process (stack...) :

- The sending Process updates the pointer (in arg_ptrs) by the value of the argument.
- The receiving Process puts this value in a local variable, the adress of which becomes the valid pointer in arg_ptrs.

d) By address arguments

The by_adress_arguments must be stored in Multics segments, which are made known to each Process through a segment number. The pointers of the by_adress_arguments in arg_ptrs

contain those segment numbers (see a)). They must be converted by TSKSTART in order to become available to the receiving Process.
The table of shared segments, noted SHSEGTABLE, give the corresponding segment number for each Process. It is accessed by all Processes.



Segno is first initialized to(-1). When a pointer refers to the shared segment of index say i, the Software makes known segno(i) for the corresponding Process.

- sending pointers (procedure pointer_send)

The pointer is translated into a structure (address_pointer), according to the pointer format a), containing MY_SEGNO.
Only MY_SEGNO is converted, while other variables remain unchanged.

The sending Process searches the pathname of the segment of number MY_SEGNO. Two cases can occur :

- if the segment was already referenced in the Process, the SHNEG_TABLE contains MY_SEGNO, and its index can be easily retreived.
- if not, the routine hcs_$fs_get_path_name is used to find the index of the segment in SHSEGTABLE. Segno is updated proceerly in the table . Finally, the segment is pointed by an index ISEG in SHSEG_TABLE, which gives the segno assigned into the receiving Process (say HIS_SEGNO). Once more, two cases can occur :
- if the segment was made known in this Process, HIS_SEGNO is valid and SEGNO is updated by HIS_SEGNO.
- if not, SEGNO is updated by -ISEG.

- Receiving pointers (procedure pointer-receive)

The pointer is read by using the structure address-pointer, containing SEGNO.

- if SEGNO > 0, then the pointer is valid.
- if SEGNO < 0, then the Process must "translate the pointer. -SEGNO gives the pathname through $HSEGTABLE. Then, the routine cv_ptr_converts the pathname into a valid

pointer. The corresponding segment number is copied in the proper storage of SHSEGTABLE.

4) Inter-Process-Communication

Processes are synchronized with help of the Multics facility Inter Process Communication IPC.

Each Process has its own channel where it receives messages. The Process-id and the channel-info are stored in SEGCOM to be shared by all Processes. They are assigned before starting execution of FORTRAN code (see crem_process, Section 4.4).

The IPC routines used are :

- ipc_$read_ev_channel
- ipc_$block
- hcs_$wakeup.

They perform exchanges of messages between the Processes. They help to realize conditional waits and to wakeup other Processes.

a) Conditional waits

The routine crem_process and crem_slave must perform unconditionnals waits. The following routines induce a conditional wait which depends on the value of an identifier ID : TSKWAIT, LOCK, EVWAIT. The test is always performed with the indivisible builtin function STAC, so as to guarantee its coherence. When the test succeeds, the Process can go on. Otherwise it has to wait.

The waiting mechanism marks the Process "waiting", in order to receive a wake message. But while this operation is performed, the waking Process may modify the identifier ID, without therefore sending a message to the not yet marked Process.

To remedy this case, the marked Process performs another test on the identifier ID (always by using the PL1 builtin function stac). But a waking message may happen to be sent in the interval, and is by the fact obsolete.

Hence it is necessary to quarantee each message to be unique, so as to identify it without ambiguity.

b) Structure of a message

A message is a two-words variable in the Multics System (see the structure event_wait_info).
In the Software, they are structured in two types :

1- mess_task
type of the message "ABORT"
"STOP"
"TSKSTART"
"TSKEND"
identifier of the task (unique by definition)

2- mess_lock_event
type of the message "UNLOCK"
"EVPOST"
identifier of the lock/event (not unique)
counter of waits (unique).

c) Receiving messages

The routines which induce waits update the variable WAITED (in SEGCOM) of the Process.
It is assigned to the corresponding waited message (that will be sent by the waking Process). Then the effective wait mechanism is performed by the procedure crem_wait, by using Inter Process Communication :

- ipc_$read_ev_chn
- ipc_$block

First of all, the procedure reads all the received message, then it remains blocked until reception of a new message. Three types of message cause the Process to cease waiting :

- a message of type "ABORT" : the procedure crem_abort is called, so as to stop the Process.
- a message of type "STOP" : it is sent by the master Process. The slave Process is free and waits for allocation of a task. This message means the normal end of the job, and the procedure returns.
- a message equal to WAITED : the procedure can return, because the received message means the end of the wait.

Before returning, the variable WAITED is updated to zero.

d) Sending messages

The following routines may send messages, by using hcs_$wakeup :

- crem_process, crem_master, crem_slave, crem_abort,
- tskstart, unlock ,evpost.

First of all, they assign the correct value to the sent message "SENT". Then they send it: either to all the Processes marked, with the variable WAITED equal to SENT, or only to the first Process of the queue in case of UNLOCK-operation.

5) Starvation problem

The LOCK library may introduce starvation if no priority is given to the waiting Processes.

```
TASK1                TASK2
SET LOCK I           TEST LOCK I
...
UNLOCK I
```

EXAMPLE OF STARVATION

To prevent this problem, the Software provides a queue mechanism. The waiting Processes (on some lock identifier ID), are partially ordered in a queue. Only the first Process is waked up when the lock ID is cleared by UNLOCK. The partial order is sufficient to guarantee the absence of starvation in the lock mechanism. The queue itself is accessed without starvation : the problem is really solved and not simply moved !

The queue is implemented with two variables and one procedure :

- a global clock shared by all Processes : CLOCK
- a local clock owned by each Process : MY-CLOCK
- a procedure get-clock

A Process which has failed in testing a lock gets into the queue by performing the procedure get-clock :

```
. PROCEDURE GET_CLOCK
. PRESENT_CLOCK = CLOCK
. MY_CLOCK = PRESENT_CLOCK + 1
. CK = STACQ(CLOCK, MY_CLOCK, PRESENT_CLOCK)
. END GET_CLOCK
```

The procedure's effect is to increment the global clock if this is not already made meanwhile. The CLOCK is updated only by means of the function STACQ so as to stay coherent. The waiting Processes are then partially ordered by their local clocks (MY_CLOCK). No loop occurs to increment the global clock, therefore no starvation is incurred.

6) Deadlock problem

The use of LOCKS and/or EVENTS may induce deadlocks where all tasks are waiting.

```
TASK 1              TASK 2
LOCK ID1            LOCK ID2
...                 ...
  LOCK ID2            LOCK ID1
  ...                 ...
  UNLOCK ID2          UNLOCK ID1
...                 ...
UNLOCK ID2          UNLOCK ID2
```

EXAMPLE OF DEADLOCK

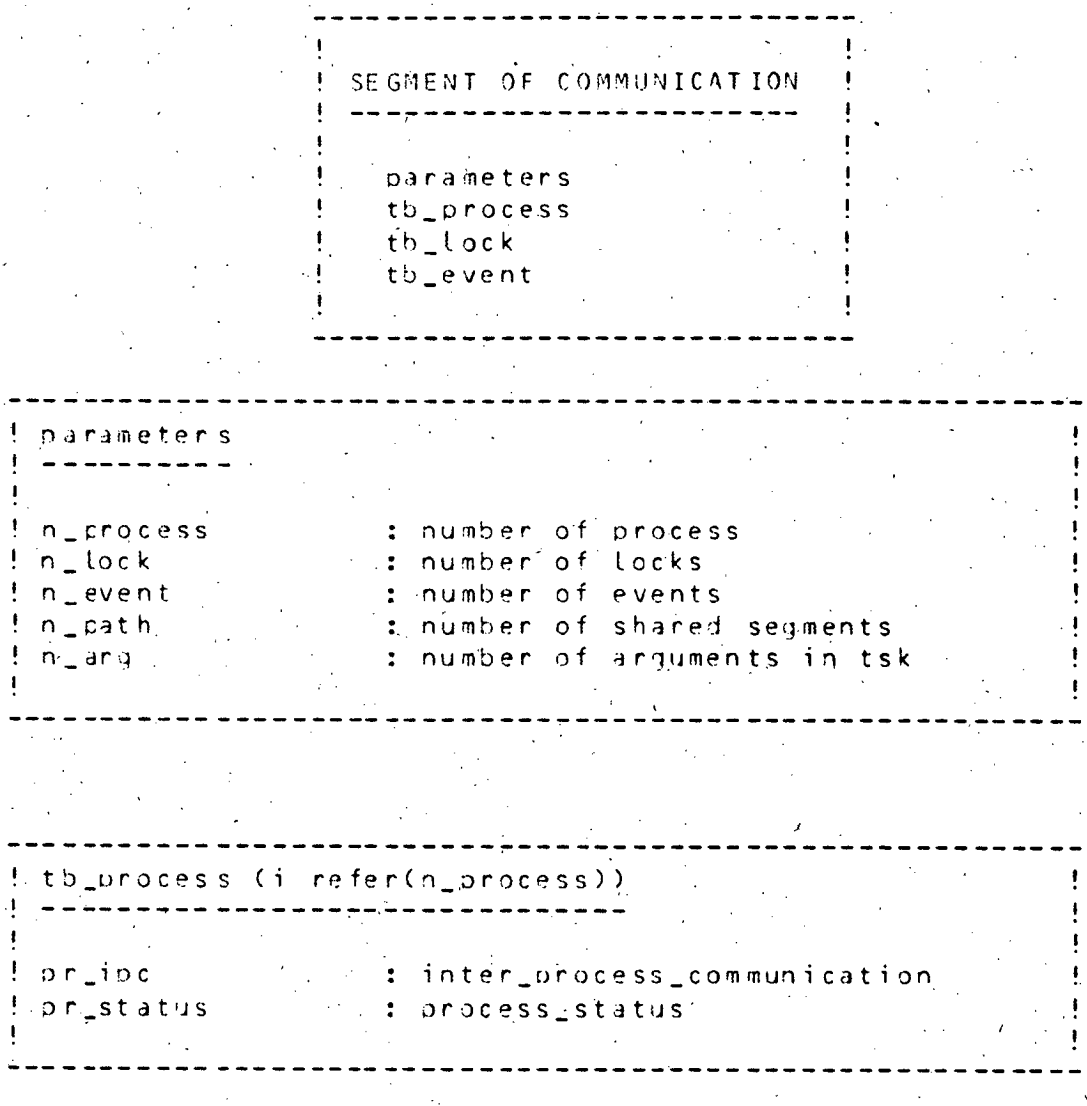
In this example, both tasks are blocked if task 1 sets the lock ID1 and task 2 sets the lock ID2.

It is the responsability of the programmer to avoid deadlocks. No detection is made by the Software. The Multics system provides a safety mechanism : Processes which remain inactive for too long a time are automatically logged out. Hence, a deadlock problem does not consume CPU time (there is no active wait loop) and is "solved" by Multics.

==========

ANNEXE 1

STRUCTURE OF COMMUNICATION SEGMENT



```
-------------------------------------------------------------
! tb_lock (i refer(n_lock))                                  !
! -------------------------                                  !
!                                                            !
! lock_assgin        : identifier of the lock                !
! lock_value         : status of the lock                    !
!                                                            !
-------------------------------------------------------------


-------------------------------------------------------------
! tb_event (i refer(n_event))                                !
! --------------------------                                 !
!                                                            !
! event_assign       : identifier of the event               !
! event_value        : status of the event                   !
!                                                            !
-------------------------------------------------------------
```

==========

ANNEXE 2

EXAMPLES OF MULTITASKING PROGRAMS

TESTS OF SYNCHRONIZATION

1) fork and join routine

```
      subroutine chef
c
c     TEST FOR FORK-AND-JOIN
c     ----------------------
c
c                    FORK        FORK
c                 !------->!  !------->!
c     TSKSTART    !         !JOIN!       !          JOIN
c     --------->!------->!====!------->!.........====!
c     !         !       !    !       !              !
c     !         !------->!    !------->!              !
c     !                                              !
c     !----------------------------------------------!
c
c
c
c     MAIL-BOX FOR SYNCHRONIZATION BETWEEN TASKS
c     ------------------------------------------
c
c     idlock  = lock-identifier   on the critical section
c     idevent = event-identifiers on the critical section
c     fliop   = flip/flop on the event to be posted
c     nwait   = number of tasks waiting
c               (they have already executed
c               the critical section)
c     nproc   = number of tasks to be synchronized
c
      common/mlbx5/idlock,idevent(2),fliop,nwait,nproc
      integer fliop
      external esclave
      external tskstart(descriptors)
      external tskwait (descriptors)
c
      dimension itask(2,10)
```

```
c
c      INITIALIZATION
c      --------------
c
         do 4 i=1,10
            itask(1,i) = 2
            itask(2,i) = 0
4        continue
c
       call mlbxinit
       print 100
       if (nproc.le.1) goto 10
c
       print 300
300    format(" nsynch=? - ntask=? - format i3")
       read 110,nsynch,ntask
110    format(i3,i3)
c
c      TSKSTART :
c          J=NUMBER OF TSKSTART - I=IDENTIFIER OF TASK
c      ------------------------------------------------
c
       do 3 j = 1,ntask
       write(11,400) j
         do 1 i=1,(nproc-1)
            call tskstart(itask(1,i),2,esclave,i,nsynch)
1        continue
c
       call esclave(nproc,nsynch)
c
c      TSKWAIT
c      -------
c
         do 2 i = 1,(nproc-1)
            call tskwait(itask(1,i))
2        continue
c
3      continue
c
       print 200
100    format(" BEGINNING OF MULTITASKING")
200    format("END OF MULTITASKING")
400    format(" TSKSTARTS:",i3)
       return
c
10     call esclave(nproc,nsynch)
```

```
return
end
```

```
c      TASK : JOIN->FORK=RANDOM_TIME->JOIN->....
c      ----------------------------------------
c
       subroutine esclave(itsk,nsynch)
c
c
c      MAIL-BOX FOR SYNCHRONIZATION BETWEEN TASKS
c      ------------------------------------------
c
c      idlock  = lock-identifier   on the critical section
c      idevent = event-identifiers on the critical section
c      fliop   = flip/flop on the event to be posted
c      nwait   = number of tasks waiting
c                (they have already executed
c                the critical section)
c      nproc   = number of tasks to be synchronized
c
       common/mlbx3/idlock,idevent(2),fliop,nwait,nproc
       integer fliop
c      MULTICS
       external random_sleep (descriptors)
c
c
       iseed = 31*itsk
         do 1 i=1,nsynch
c                      -- fork = random_time
       call random_sleep(iseed)
c                      -- join = write in shared file
       write(11,100) itsk,i
       call forkjoin
1        continue
       return
100      format(" TASK:",i3," SYNCHRO:",i3)
c
       end
```

```
      subroutine forkjoin
c
c     FORK AND JOIN OF NPROC TACHES
c     -----------------------------
c
c     ALL TASKS MUST EXECUTE THIS CRITICAL SECTION
c     THEY ARE WAITING UNTIL THE LAST ONE POST AN EVENT
c
      %include mlbx ;
c
c     MAIL-BOX FOR SYNCHRONIZATION BETWEEN TASKS
c     ------------------------------------------
c
c     idlock  = lock-identifier   on the critical section
c     idevent = event-identifiers on the critical section
c     fliop   = flip/flop on the event to be posted
c     nwait   = number of tasks waiting
c               (they have already executed
c                the critical section)
c     nproc   = number of tasks to be synchronized
c
      common/mlbx$/idlock,idevent(2),fliop,nwait,nproc
      integer fliop
c     MULTICS
      external lockon  (descriptors)
      external lockoff (descriptors)
      external evpost  (descriptors)
      external evwait  (descriptors)
      external evclear (descriptors)
      external evtest  (descriptors)
c     FIN MULTICS
      logical  evtest
c
c
      call lockon (idlock)
c                     -- enter into the Critical Section
        nwait = nwait + 1
c                     --
        if (nwait .ge. nproc) goto 10
c                     -- nwait < nproc => wait for the others
      locflp = fliop
      call lockoff (idlock)
c                     -- exit from the Critical Section
      call evwait(idevent(locflp))
      return
c
c                     -- nwait = nproc => wakeup the others
```

```
10      nwait = 0
        call evpost(idevent(fliop))

c                       -- change the flipflop
        flicp = fliop + 1
        if (fliop.eq.3)  fliop = 1
        if (evtest (idevent (fliop)))
     S       call evclear(idevent(fliop))
        call lockoff(idlock)
c                       -- exit from the Critical Section
        return
c
        end
c
c
        subroutine mlbxinit
c
c       INITIALIZATION OF MLBX
c       ----------------------
c
c
c       MAIL-BOX FOR SYNCHRONIZATION BETWEEN TASKS
c       ------------------------------------------
c
c       idlock  = lock-identifier   on the critical section
c       idevent = event-identifiers on the critical section
c       fliop   = flip/flop on the event to be posted
c       nwait   = number of tasks waiting
c                 (they have already executed
c                  the critical section)
c       nproc   = number of tasks to be synchronized
c
        common/mlbx$/idlock,idevent(2),fliop,nwait,nproc
        integer fliop
        external lockasgn(descriptors)
        external evasgn  (descriptors)
c
        call lockasgn(idlock)
        call evasgn(idevent(1))
        call evasgn(idevent(2))
c
        nwait = 0
        fliop = 1
        print 300
        read  310 ,nproc
300     format ("nproc = ? - format i3")
310     format (i3)
```

```
return
end
```

master.absin

```
cwd crem>synch
cl //*.absout -bf
cl synchro -bf
atsh 11 synchro
&attach
crem_init seg
o
mlbx .
ear // -q 0
ear //1 -q 0
ear //2 -q 0
crem_process seg chef
  4
  2  5
det 11
```

//.absin

```
hmu
cwd crem>synch
atsh 11 synchro
cat
crem_process seg
```

master.absout

```
Absentee user Erhel Menusin logged in: 03/31/83  1308.5 hfe Thu
r 13:08 2.056 36

cwd crem>synch
r 13:08 0.074 0

cl //*.absout -bf
r 13:08 0.138 1

dl synchro -bf
r 13:08 0.097 0

atsh 11 synchro
io_call: No I/O switch, file11
r 13:08 0.517 16

crem_init seg

 PARAMETRES PAR DEFAUT :
n_process=p=2  n_lock=l=30  n_event=e=30  n_arg=a=30
n_shseg=s=50
underflow=u=0  dump=d=0
 CHANGER LES PARAMETRES ?
 repondre par oui,non, ou :<pathname>
c

p=* l=* e=* a=* s=* u=0/1 d=0/1 ?
p=4;
p=4;

 PATHNAMES DES SEGMENTS PARTAGES ?
 terminer la liste par un blanc suivi d'un point
mlbx .

r 13:08 0.353 12

ear // -q 0
ID: 110843.2; 1 already requested.
r 13:08 0.896 15

ear //1 -q 0
ID: 110845.2; 2 already requested.
r 13:08 0.152 1
```

```
ear //2 -q C
ID: 110845.6; 3 already requested.
r 13:08 0.152 2

crem_process seg_chef

PROCESS MASTER NUMERO    4

nproc = ? - format i3
  4
 BEGINNING OF MULTITASKING
 nsynch=? - ntask=? - format i3
  5  2
END OF MULTITASKING


INTER_PROCESS_COMMUNICATION
nombre de messages recus    :                    5
nombre de messages envoyes :                    27
r 13:09 1.509 40

det 11
r 13:09 0.054 0


Absentee user Erhel Menusin logged out 03/31/83  1309.1 hfe Thu
CPU usage 6 sec, memory usage 2.0 units
```

```
//.absout
---------


Absentee user Erhel Menusin logged in: 03/31/83  1308.7 hfe Thu
r 13:08 2.118 22

hmu

Multics MR9.1.12, load 44.0/110.0; 44 users, 36 interactive, 4 daemons.
Absentee users 4/9
r 13:08 0.086 1

cwd crem>synch
r 13:08 0.060 0

atsh 11 synchro
io_call: No I/O switch. file11
r 13:08 0.528 3

pat
error_output          syn_ user_i/o
     -inh close get_line get_chars
user_input            syn_ user_i/o
     -inh close put_chars
user_i/o              abs_io_
     ">user_dir_dir>Menusin>Erhel>crem>synch>//.absin"
     stream_input_output
user_output           syn_ user_i/o
     -inh close get_line get_chars
file11
     vfile_ >udd>Menusin>Erhel>crem>synch>synchro
     -append -no_end -blocked 100 -share 100
          (not open)
r 13:08 0.147 0

crem_process seg

PROCESS SLAVE NUMERO   1

INTER_PROCESS_COMMUNICATION
nombre de messages recus    :                    12
nombre de messages envoyes :                     11
r 13:09 1.463 19
```

```
Absentee user Erhel Menusin logged out 03/31/83  1309.1 hfe Thu
CPU usage 4 sec, memory usage 0.0 units
```

shared file synchro

```
TSK STARTS:  1
TASK:  1 SYNCHRO:  1
TASK:  2 SYNCHRO:  1
TASK:  4 SYNCHRO:  1
TASK:  3 SYNCHRO:  1
TASK:  1 SYNCHRO:  2
TASK:  2 SYNCHRO:  2
TASK:  4 SYNCHRO:  2
TASK:  3 SYNCHRO:  2
TASK:  1 SYNCHRO:  3
TASK:  2 SYNCHRO:  3
TASK:  3 SYNCHRO:  3
TASK:  4 SYNCHRO:  3
TASK:  1 SYNCHRO:  4
TASK:  2 SYNCHRO:  4
TASK:  3 SYNCHRO:  4
TASK:  4 SYNCHRO:  4
TASK:  4 SYNCHRO:  5
TASK:  3 SYNCHRO:  5
TASK:  2 SYNCHRO:  5
TASK:  1 SYNCHRO:  5
TSK STARTS:  2
TASK:  1 SYNCHRO:  1
TASK:  2 SYNCHRO:  1
TASK:  3 SYNCHRO:  1
TASK:  4 SYNCHRO:  1
TASK:  4 SYNCHRO:  2
TASK:  2 SYNCHRO:  2
TASK:  3 SYNCHRO:  2
TASK:  1 SYNCHRO:  2
TASK:  1 SYNCHRO:  3
TASK:  2 SYNCHRO:  3
TASK:  3 SYNCHRO:  3
TASK:  4 SYNCHRO:  3
TASK:  1 SYNCHRO:  4
TASK:  2 SYNCHRO:  4
TASK:  3 SYNCHRO:  4
TASK:  4 SYNCHRO:  4
TASK:  4 SYNCHRO:  5
TASK:  3 SYNCHRO:  5
TASK:  2 SYNCHRO:  5
TASK:  1 SYNCHRO:  5
```

2) rendez_vous routines

```
c TEST FOR THE RENDEZ-VOUS SUBROUTINES WAKEUP AND WAIT
c ----------------------------------------------------
c
      subroutine rvme
      common/mbrv$/nrv,nwake,nwait,mbox(2,2)
c     nrv = total number of rendez-vous
c     nwake = number of wakeup
c     nwait = number of wait
c     mbox  = mail-box for the rendez-vous
c     mbox(1,i) = waited by receiver i
c     mbox(2,i) = posted back by receiver i
c
      external tskstart      (descriptors)
      external tskwait       (descriptors)
      external random_sleep  (descriptors)
      external rvother       (descriptors)
      integer other
      dimension itsk(2)
c
      me = 1
      other = 2
      iseed = 13
      call rvinit (2,mbox)
      print 100
100   format(" nrv - nwake - nwait ?? format 3*i4")
      read  110,nrv,nwake,nwait
110   format(3 (i4))
      itsk(1)=2
      call tskstart ( itsk,2,rvother,other,me)
c
        do 1 i=1,nrv
           do 2 j=1,nwake
              call random_sleep(iseed)
              call wakeup (other,mbox)
              write(11,200) me,i,j
2          continue
           do 3 j=1,nwait
              call random_sleep(iseed)
              write(11,300) me,i,j
              call wait(me,mbox)
3          continue
1       continue
c
```

```
      call tskwait(itsk)
c
200   format("TASK",i2," RENDEZ-VOUS",i3," WAKEUP",i3)
300   format("TASK",i2," RENDEZ-VOUS",i3," WAIT  ",i3)
      return
      end
```

```
      subroutine rvother (me,other)
c
      common/mbrv$/nrv,nwake,nwait,mbox(2,2)
c
      iseed = 7
        do 1 i=1,nrv
           do 2 j=1,nwake
              call random_sleep(iseed)
              write(11,200) me,i,j
              call wait (me,mbox)
2          continue
           do 3 j=1,nwait
              call random_sleep(iseed)
              call wakeup(other,mbox)
              write(11,300) me,i,j
3          continue
1       continue
c
200   format("TASK",i2," RENDEZ-VOUS",i3," WAIT  ",i3)
300   format("TASK",i2," RENDEZ-VOUS",i3," WAKEUP",i3)
      return
      end
```

```
c      RENDEZ-VOUS BETWEEN TWO TASKS
c      -----------------------------
c
c      one task is sender and the other is receiver
c      both emission and reception are bloquant :
c        the sender   must wait for the previous reception
c        the receiver must wait for the actual   emission
c
c      SUBRCUTINE WAKEUP                    SUBROUTINE WAIT
c      -----------------                    ---------------
c
c      PARAMETERS (IN):                     PARAMETERS (IN):
c      ----------------                     ----------------
c
c      - other = number of the receiver     - me   = my number
c      - mbrv  = mail-box for the rendezvs  - mbrv = mail-box
c      - mbrv(1,i) = event posted by sender
c                    and waited by receiver i
c      - mbrv(2,i) = back event posted by receiver i
c                    and waited by sender
c
c      ALGORITHM:                           ALGORITHM :
c      ---------                            ---------
c
c      - wait for back mbrv(2,other)         - wait for in
c                                                 mbrv(1,me)
c      - post the in   mbrv(1,other)      - post the back
c                                                 mbrv(2,me)
c-------------------------------------------------------------
c
      subroutine wakeup (other,mbrv)
      dimension  mbrv(2,1)
      integer    other
      external   evpost   (descriptors)
      external   evclear  (descriptors)
      external   evwait   (descriptors)
c
c                    -- wait for the previous back
      call evwait (mbrv(2,other))
      call evclear(mbrv(2,other))
c                    -- post the sent
      call evpost (mbrv(1,other))
c
      return
      end
c
      subroutine wait(me,mbrv)
```

```
      cimension   mbrv(2,1)
      integer     me
      external    evpost    (descriptors)
      external    evclear   (descriptors)
      external    evwait    (descriptors)
c
c                       -- wait for the emission
      call evwait (mbrv(1,me))
      call evclear(mbrv(1,me))
c                       -- post the back
      call evpost (mbrv(2,me))
c
      return
      end
c
      subroutine rvinit(ntask,mbrv)
c
c     parameter ntask = number of tasks
      cimension   mbrv(2,ntask)
      external    evpost    (descriptors)
      external    evasgn    (descriptors)
c
c                       -- assign the events
c                       -- and post the backs
        do 1 i=1,ntask
           call evasgn(mbrv(1,i))
           call evasgn(mbrv(2,i))
           call evpost(mbrv(2,i))
1       continue
c
      return
      end
```

shared file RENDEZ_VOUS

```
TASK 1  RENDEZ-VOUS   1  WAKEUP   1
TASK 2  RENDEZ-VOUS   1  WAIT     1
TASK 2  RENDEZ-VOUS   1  WAIT     2
TASK 1  RENDEZ-VOUS   1  WAKEUP   2
TASK 1  RENDEZ-VOUS   1  WAKEUP   3
TASK 2  RENDEZ-VOUS   1  WAIT     3
TASK 1  RENDEZ-VOUS   1  WAKEUP   4
TASK 2  RENDEZ-VOUS   1  WAIT     4
TASK 1  RENDEZ-VOUS   1  WAKEUP   5
TASK 1  RENDEZ-VOUS   1  WAIT     1
TASK 2  RENDEZ-VOUS   1  WAIT     5
TASK 2  RENDEZ-VOUS   1  WAKEUP   1
TASK 1  RENDEZ-VOUS   1  WAIT     2
TASK 2  RENDEZ-VOUS   1  WAKEUP   2
TASK 1  RENDEZ-VOUS   1  WAIT     3
TASK 2  RENDEZ-VOUS   1  WAKEUP   3
TASK 1  RENDEZ-VOUS   2  WAKEUP   1
TASK 2  RENDEZ-VOUS   2  WAIT     1
TASK 1  RENDEZ-VOUS   2  WAKEUP   2
TASK 2  RENDEZ-VOUS   2  WAIT     2
TASK 1  RENDEZ-VOUS   2  WAKEUP   3
TASK 2  RENDEZ-VOUS   2  WAIT     3
TASK 1  RENDEZ-VOUS   2  WAKEUP   4
TASK 2  RENDEZ-VOUS   2  WAIT     4
TASK 1  RENDEZ-VOUS   2  WAKEUP   5
TASK 2  RENDEZ-VOUS   2  WAIT     5
TASK 1  RENDEZ-VOUS   2  WAIT     1
TASK 2  RENDEZ-VOUS   2  WAKEUP   1
TASK 2  RENDEZ-VOUS   2  WAKEUP   2
TASK 1  RENDEZ-VOUS   2  WAIT     2
TASK 1  RENDEZ-VOUS   2  WAIT     3
TASK 2  RENDEZ-VOUS   2  WAKEUP   3
```

==========

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique