Application Interfaces to BPFS: a Basic Parallel File System.

Robert D. Russell

Laboratoire de l'Informatique du Parallélisme

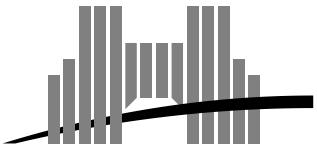
École Normale Supérieure de Lyon
Unité de recherche associée au CNRS n° 1398

Application Interfaces to BPFS: a Basic Parallel File System

Robert D. Russell

June 1998

Research Report N° 98-28



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.00
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

Application Interfaces to BPFS: a Basic Parallel File System

Robert D. Russell

June 1998

Abstract

This report describes three application program interfaces to BPFS, a distributed, modular parallel file system designed for use on clusters of workstations. These interfaces are called API0, CLI, and MPI-IO.

API0 is the first of an anticipated series of low-level, experimental client interfaces to BPFS. It is an "unconventional" interface in many respects: it is not particularly "UNIX-like", it is block-oriented rather than byte oriented, it reads and writes system buffers as well as user-defined data areas, and it is asynchronous. It also provides time-regulated "data streaming" operations and user-level control of both server-side caching and per-file striping onto disks.

Although API0 can be used directly from a user application program, it can also be used "under" a more conventional interface, as has been done for the next two interfaces.

CLI is a "C Library Interface" implemented on top of API0 that exactly mimics the Standard C I/O library interface, but accesses parallel files stored by BPFS rather than sequential files stored by the host file system.

The third interface is the ROMIO version of the standard MPI-IO interface which has been implemented on top of API0 to support access to BPFS files from parallel programs that use the Message Passing Interface (MPI).

Keywords: parallel file systems, distributed file systems, file systems, application program interfaces.

Résumé

Ce rapport décrit trois interfaces de programmation de BPFS, un système de fichiers distribué modulaire conçu pour des grappes de stations de travail. Ces interfaces se nomment respectivement API0, CLI et MPI-IO.

API0 est la première d'une série d'interfaces d'accès à BPFS de bas niveau. Cette interface est originale à plusieurs titres : elle n'obéit pas à la philosophie classique des fichiers sous UNIX, elle opère en mode bloc et non en mode caractère, elle permet la lecture/écriture de tampons "systèmes" et de données utilisateurs et enfin elle est asynchrone. De plus, des opérations de flux de donnée periodique ansi que le paramétrage des tampons du coté serveurs par les clients sont disponibles.

Bien que l'interface API0 puisse être utilisée directement par n'importe quelle application, deux interfaces de niveau supérieur ont été définies pour une utilisation plus aisée.

CLI est une interface s'appuyant sur API0 qui fournit les primitives standards d'entrée/sortie de la "libc". Ces primitives accèdent aux fichiers parallèles gérés par BPFS et non aux fichiers séquentiels UNIX traditionnels.

La troiseme interface est une version de l'interface ROMIO (elle même sous-ensemble de l'interface standard MPI-IO) implantée au-dessus d'API0. Cette interface permet donc aux applications développées au-dessus de MPI de s'executer sans modification au-dessus de BPFS.

Mots-clés: système de fichiers parallèle, système de fichiers distribué, système de fichiers, interfaces de programmation.

Application Interfaces to BPFS: a Basic Parallel File System

Robert D. Russell

2nd June 1998

Abstract

This report describes three application program interfaces to BPFS, a distributed, modular parallel file system designed for use on clusters of workstations. These interfaces are called API0, CLI, and MPI-IO.

API0 is the first of an anticipated series of low-level, experimental client interfaces to BPFS. It is an "unconventional" interface in many respects: it is not particularly "UNIX-like", it is block-oriented rather than byte oriented, it reads and writes system buffers as well as user-defined data areas, and it is asynchronous. It also provides time-regulated "data streaming" operations and user-level control of both server-side caching and per-file striping onto disks.

Although API0 can be used directly from a user application program, it can also be used "under" a more conventional interface, as has been done for the next two interfaces.

CLI is a "C Library Interface" implemented on top of API0 that exactly mimics the Standard C I/O library interface, but accesses parallel files stored by BPFS rather than sequential files stored by the host file system.

The third interface is the ROMIO version of the standard MPI-IO interface which has been implemented on top of API0 to support access to BPFS files from parallel programs that use the Message Passing Interface (MPI).

Keywords: parallel file systems, application program interface

1 Introduction

This report describes three application program interfaces to BPFS — a Basic Parallel File System [6]. These interfaces are called API0, CLI, and MPI-IO.

API0 is the first of an anticipated set of experimental, low-level client interfaces to BPFS. It is an "unconventional" interface in many respects:

- it is not particularly "UNIX-like",
- it is block-oriented rather than byte oriented,
- it reads and writes system buffers as well as user-defined data areas,
- it has no notion of "current position" within a file, requiring a user to indicate a specific block number for every I/O operation,
- it is asynchronous in that functions generally come in pairs — the first in the pair to "start" an operation, the second in the pair to "wait-for" the completion of the operation,
- its functions all return NULL pointers on correct operation, pointers to a buffer containing an error message on incorrect operation,
- it provides no communication or synchronization between clients,
- it provides time-regulated "data streaming" operations as well as more conventional request-reply operations.
- it provides user control over per-file server-side caching.
- it provides user control over per-file striping onto disks.

Although API0 can be used directly from a user application program, it can also be used "under" a more conventional interface. This is almost certainly necessary if it is to be used by parallel processes, since API0 itself makes no attempt to coordinate actions between clients. Indeed, it has no notion of multiple clients.

CLI is a "C Library Interface" implemented on top of API0 that exactly mimics the Standard C I/O library interface but accesses BPFS rather than the host file system. This interface is designed for high-level applications that wish to access the parallel file system. These applications can be parallel programs as well as non-parallel programs, such as servers, that wish to access the parallel file system because of its greater bandwidth and lower latency than the normal host file system. An example might be a video display program.

MPI-IO is the ROMIO version of the standard MPI-IO interface which has been implemented on top of API0 to support access to BPFS files from parallel programs that use the Message Passing Interface (MPI) [5].

API0 directly interfaces to BPFS, and the other two interfaces are implemented "on top" of API0, as shown in Figure 1.

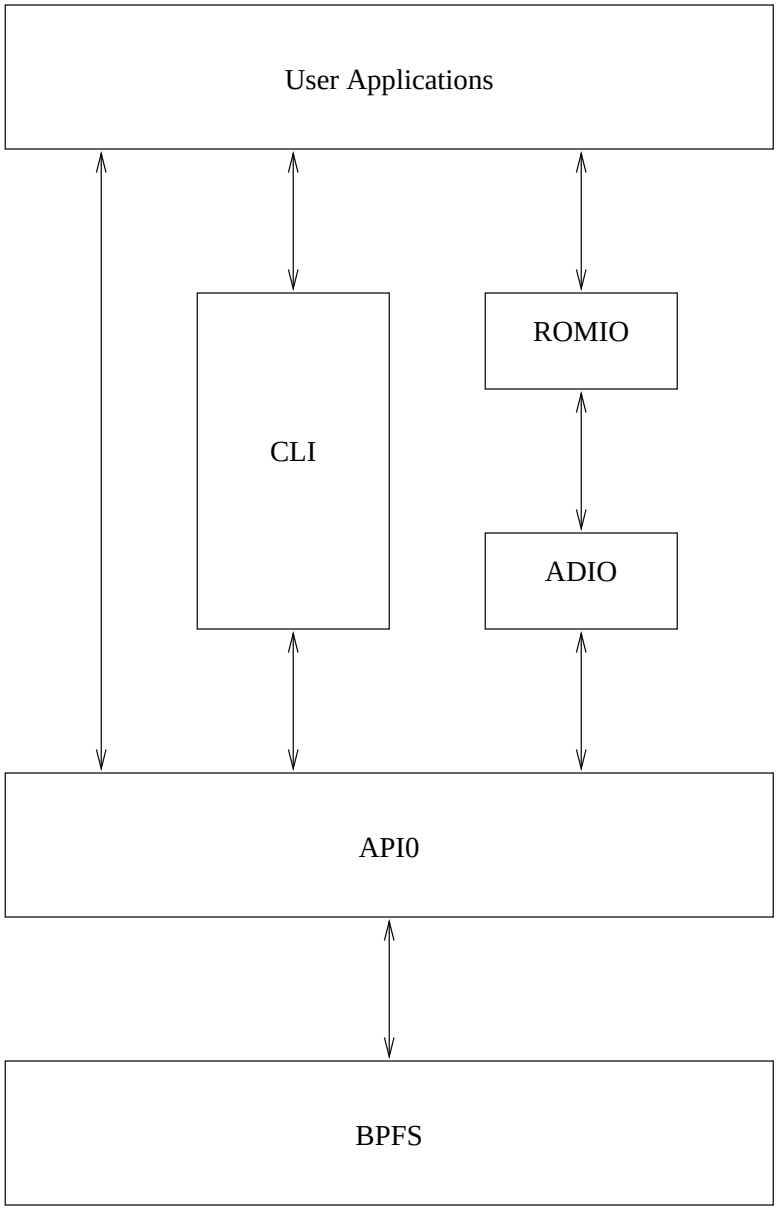


Figure 1: Layering of the API0, CLI, and MPI-IO interfaces to BPFS.

This report is organized as follows. Section 2 describes the API0 interface to BPFS, Section 3 describes the C library interface to BPFS, and Section 4 describes the MPI-IO interface to BPFS. The Appendix has two sections containing alphabetical listings of the prototypes of all the functions in API0 and CLI.

2 API0

API0 is a low-level interface intended to provide a programming interface to the facilities provided by a BPFS client. It therefore reflects many of the design features found in BPFS. For example, it performs I/O in units of fixed-size blocks, it is highly asynchronous, and it uses "locations" to refer to network addresses.

All the symbols, data structures, and function prototypes for API0 are defined in the header file "api0.h", which must be included by any program wishing to utilize this interface.

There are a number of conventions that have been followed in the design of the API0 interface, which contains 65 functions:

- All 65 functions have names starting with "**bpfs_**".
- All but two of the 65 functions have type "**char ***". If a function of this type is able to perform properly, it returns the value NULL. If a function of this type detects an error, it returns a pointer to a buffer containing the text of an explanatory error message represented as a null-terminated C string. The two exceptions to this rule are the functions **bpfs_blocksize** and **bpfs_perror**, which always succeed.
- Most functions come in pairs. The first in the pair is given a name starting with "**bpfs_start_**" and is called to initiate an action — control returns to the calling program as soon as the action is started. The second in the pair is given a name starting with "**bpfs_wait_for_**" and is called to wait indefinitely for the completion of an action. If the "**bpfs_start_**" function in a pair returns an error, the action is not started and the corresponding "**bpfs_wait_for_**" function in the pair should not be called.
- For convenience, "wrapper" functions are provided that call the two functions in a pair one right after the other. If the pair is called **bpfs_start_xxx** and **bpfs_wait_for_xxx**, then the wrapper function

is called **bpfs_xxx**. These wrapper functions will not be mentioned again in this section of the report. However, their prototypes are listed in the appendix with all the other API0 functions.

There are 18 "start_ – wait_for_" function pairs, which together with the 9 corresponding wrapper functions account for 54 out of the total of 65 functions defined in API0.

2.1 Initialization and Termination

Any program utilizing API0 must call the function **bpfs_setup** before doing anything else. When the program is finished using API0 functions, it may call **bpfs_endup** prior to termination. However, **bpfs_setup** utilizes the "atexit" facility of standard C to ensure that **bpfs_endup** is automatically called whenever the application program calls **exit** or returns from **main**.

```
extern char *bpfs_setup( unsigned int dump_stuff );

extern char *bpfs_endup( void );
```

bpfs_setup takes as an input parameter **dump_stuff**, a flag which should have a non-zero value if the caller wishes to print out some statistics relating to buffer usage at the end of execution. If this flag has the value zero, nothing is printed. If the value returned by **bpfs_setup** is not NULL, API0 was unable to initialize itself properly, and the calling program should make no attempt to utilize any other API0 functions except **bpfs_perror** to print the error message returned.

2.2 BLOCKSIZE

API0 performs all its data I/O in blocks of a fixed size. This value is fixed by the underlying BPFS implementation, and is constant throughout the parallel file system. In this report this size will be indicated by the symbol BLOCKSIZE. This value is returned as the result of the following API0 function:

```
extern unsigned int bpfs_blocksize( void );
```

This function always succeeds.

2.3 Buffers

Many of the functions defined in API0 utilize system-defined buffers. Each buffer is the same fixed size and holds BLOCKSIZE bytes.

Two functions are provided to obtain and release buffers:

```
extern char *bpfs_get_buffer( char **buf );

extern char *bpfs_return_buffer( char *buf );
```

bpfs_get_buffer returns in its output parameter **buf** the address of a newly allocated system buffer containing BLOCKSIZE bytes. If no such buffer can be allocated, **bpfs_get_buffer** returns an error message and NULL is returned as the value of **buf**.

bpfs_return_buffer takes as its input parameter **buf** the address of a previously allocated system buffer and returns it to the pool of free buffers for eventual reallocation. If the value pointed to by **buf** is not a valid system buffer, **bpfs_return_buffer** returns an error message.

2.4 Error Messages

As stated above, all but two functions in API0 return the value NULL when they work correctly, and a pointer to a system buffer containing the text of an error message when they don't work correctly. The reason for returning error messages rather than error codes is quite simple: these messages may be generated on different nodes in the file system (the node on which API0 itself is running, the node on which the manager is running, the nodes on which the servers are running), and each of these nodes may have a different operating system which of course implies possibly different error codes for the same error. Therefore, to have BPFS functions return meaningful error codes, we would have to define a "universal" error code set, and have every node map all its error codes into that universal set before sending them to other nodes in the network. Instead, BPFS uses another alternative — have the node which generates the error immediately convert its local error code into a text message, and send that message to other nodes in the network. The disadvantage of this technique is that it is more difficult to write application code to test for a certain text error message than to test for a certain numeric error code. However, since the vast majority of numeric error codes are used by application programs solely to look up an appropriate error message and print it, this objection does not appear to be a severe limitation.

The programmer can, of course, do anything he likes with this error message, but in order to conserve system resources, the buffer should be returned

to the free pool by calling **bpfs_return_buffer** after the programmer has finished with it. The programmer should not use the system routine **free** to release error message buffers, since these buffers are part of a pool that is managed internally by API0.

There is one function provided to print out the error message on the standard error device and then return the buffer to the free pool:

```
extern void bpfs_perror( const char *info, char *buf );
```

bpfs_perror takes two input parameters: **info**, which is the address of a null-terminated C string, and **buf**, which is the address of a buffer containing an error message as returned by an API0 function. This routine will print these two strings on a single line on the standard error device in the form:

info: buf

However, if **info** is NULL, only the string **buf** is printed. After printing, **bpfs_perror** returns **buf** to the free buffer pool. If **buf** is NULL, nothing is printed or returned to the free pool.

This function always succeeds.

2.5 PFILE descriptors

The standard sequence of operations necessary to process the data in a parallel file is the same as those needed to access a sequential file:

1. open the file,
2. repeatedly read data from and/or write data to the file,
3. close the file.

When the file is opened, API0 will return to the calling program a "PFILE" descriptor that the program must use in all subsequent read, write and close operations on that file. The symbol PFILE stands for "Parallel File", and is analogous to the "FILE" descriptor utilized in the Standard C I/O library. Unlike the C library, however, a PFILE descriptor is also used in API0 as a handle to link many "start_ – wait_for_" function pairs. As far as the programmer is concerned, a PFILE descriptor is an opaque object whose only use is as a parameter to many of the API0 functions.

2.6 Opening a File

Before data can be read from or written to a parallel file the program must successfully open the file using the following pair of functions:

```
extern char *bpfs_start_open( const char *file_name,
        unsigned int mode,
        bpfs_open_attributes_t *attr,
        PFILE **pfptr );

extern char *bpfs_wait_for_open( PFILE *pfptr );
```

Note that **bpfs_start_open** can be called many times to start opening many different files before any calls are made to **bpfs_wait_for_open**, but **bpfs_wait_for_open** must be the next operation performed on the **pfptr** parameter returned by each **bpfs_start_open** call. In other words, no read or write operations can be started on a file until **bpfs_wait_for_open** has been called to ensure that the file was in fact opened properly.

The parameters to **bpfs_start_open** are as follows:

- **file_name** — an input parameter containing the name of the file the calling program wishes to open, represented as a null-terminated C string. The form of this name must follow the conventions of the underlying file system.

- **mode** — an input parameter containing a value indicating how the calling program wishes to open this file. This value must be one of the three values BPFS_READ_ONLY, BPFS_WRITE_ONLY, and BPFS_READ_WRITE, depending on whether the program wishes to read, write, or read and write the file. When a file is opened with BPFS_READ_ONLY, it is considered to be open in "shared, read-only" mode, which means that the only other processes allowed to simultaneously access this file must also use the same mode. When a file is opened with BPFS_WRITE_ONLY, it is considered to be open in "exclusive, write-only" mode, which means that no other process is allowed to simultaneously access this file in any mode. When a file is opened with BPFS_READ_WRITE, it is considered to be open in "shared, update" mode, which means that the only other processes allowed to simultaneously access this file must also use the same mode.

- **attr** — an input parameter containing a pointer to a structure which the calling program has filled in with values that will define the attributes used to access this file. This structure is defined in the section

below. If this parameter is NULL, a set of default attributes will be used.

- **pfptr** — an output parameter into which this routine stores the address of a PFILE descriptor for a successful open. If **bpfs_start_open** returns an error message, the value of **pfptr** is undefined.

2.6.1 Open Attributes

When a file is opened, there are a number of options that define how the file will be accessed. In addition, when a parallel file is created, there are a large number of attributes that must be provided in order to properly define the new file. Because there are so many of these options and attributes, they are not passed as separate parameters to the open function, but rather are stored as fields in a structure and a pointer to that structure is passed to the open. This structure has the type "**bpfs_open_attributes_t**", and contains the following fields:

- **open_truncate** — can have values 0 or 1. If it is set to 1, an existing file opened in mode BPFS_WRITE_ONLY will be truncated at the time it is opened. Truncation means that all data in the file is destroyed, and the size of the file becomes 0. If this field is set to 0, an existing file opened in mode BPFS_WRITE_ONLY retains any existing data, although this data may be subsequently be overwritten by writes from the program. If the file is opened in BPFS_READ_ONLY or BPFS_READ_WRITE modes, the value of this field is ignored.

- **open_create** — can have values 0, 1 or 2. If it is set to 0, a file opened in modes BPFS_WRITE_ONLY or BPFS_READ_WRITE must exist at the time of the open, and the open will utilize that file with its existing attributes (the data will also be truncated if **open_truncate** is set to 1 and the mode is BPFS_WRITE_ONLY). If it is set to 1, a file opened in those modes must not exist at the time of the open, and a new file will be created by the open. If it is set to 2, an new file will be created by an open in those modes only if necessary (i.e., it does not already exist). A newly created file will utilize the attributes defined by the remaining fields in this structure. If the file is opened in BPFS_READ_ONLY mode, the value of this field is ignored because the file must already exist prior to the open.

- **open_no_reply_on_write** — can have values 0 or 1. If it is set to 1, buffered write operations do not expect a reply from the file server

confirming the number of bytes written to the file. If it is set to 0, a buffered write operation is not complete until the file server has sent back the number of bytes written to the file by the write operation. This value is ignored on direct write operations, because they always complete only after all blocks have been sent to the server.

- **open_protection** — if this open causes the creation of a new file, that file will be given the access rights encoded in this field. The exact form of these rights depends on the underlying file system. If this is a UNIX system, these rights take the form of an octal number containing 3 digits. The left-most digit represents the file owner's access rights, the middle digit represents the file group's access rights, and the right-most digit represents the access rights for everyone else (the rest of the world). Within each octal digit, the left-most bit is set to 1 to allow read access, the middle bit is set to 1 to allow write access, and the right-most bit is set to 1 to allow execute access.

- **open_mapping** — if this open causes the creation of a new file, that file will be given the mapping function represented by the number in this field. A value of 0 means to use the implementation-defined default mapping function. A value of 1 means use regular striping with uniform thickness. A value of 2 means to use pseudo-random striping with uniform thickness. Other values refer to mapping functions which must be provided by the program in an implementation-defined manner.

- **open_thickness** — if this open causes the creation of a new file, that file will be given the striping thickness given by the number in this field. The thickness is the number of consecutive blocks per server in each stripe across the disks. A value of 0 means the default thickness (1).

- **open_nservers** — if this open causes the creation of a new file, that file will be given the number of servers given by the number in this field. A value of 0 means the default number, which is the number of servers contained in the field **open_server_list** if that list is not empty, otherwise it is the number of servers currently available to the manager of this file.

- **open_server_list** — if this open causes the creation of a new file, the data in that file will be stored on the servers given in this list (if the list is not empty or NULL). This field is a null-terminated C string containing a comma- or blank-separated list of server locations in the

external representation defined by the underlying network layer. If this list is empty or NULL, the servers for this file will be all those currently available to the manager of this file.

The symbol BPFS_OPEN_ATTRIBUTES_INITIALIZER has been defined in the header file "api0.h" as a convenience to the programmer for use in initializing a newly declared structure of this type with reasonable default values (see below). This symbol could be used in a declaration of the variable **xyz** as follows:

```
bpfs_open_attributes_t  xyz = BPFS_OPEN_ATTRIBUTES_INITIALIZER;
```

If this structure is not modified after initialization and its address is passed as the **attr** parameter to **bpfs_start_open**, then the open attributes will have the same default values as when the NULL pointer is passed as the **attr** parameter to **bpfs_start_open**. These values are:

Field	Value	Meaning
open_truncate	0	do not truncate
open_create	2	create if necessary
open_no_reply_on_write	1	do not reply to buffered writes
open_protection	0666	read and write access to all
open_mapping	0	implementation-defined default (1)
open_thickness	0	implementation-defined default (1)
open_nservers	0	all available to manager
open_server_list	NULL	use servers available to manager

2.7 Obtaining Information about an Open File

Once a file has been opened successfully, the programmer can obtain information about it by calling the following function:

```
extern char *bpfs_open_information( PFILE *pfptr,
        char **file_name,
        unsigned int *mode,
        unsigned int *mapping,
        unsigned int *thickness,
        unsigned int *nservers,
        bpfs_server_status_t **buf );
```

Note that this function is not part of a "start_ – wait_for_" pair, since the information it returns is immediately available from the open file control block pointed to by the input parameter **pfptr**. All the other parameters are output parameters, and are the addresses of locations into which this function will store the indicated results. If any particular return value is not wanted, the corresponding parameter should be NULL. The results stored in non-NULL parameters are as follows:

- **file_name** — the name by which the file was opened. This is stored as a null-terminated C string in a system buffer, and it is the address of this system buffer that is returned in **file_name**. After the program is finished using this information, this buffer should be returned by calling **bpfs_return_buffer**.
- **mode** — the mode by which the file was opened. This will be one of BPFS_READ_ONLY, BPFS_WRITE_ONLY, or BPFS_READ_WRITE.
- **mapping** — the number of the striping function associated with the file.
- **thickness** — the striping thickness associated with the file.
- **nservers** — the number of servers across which the file is striped.
- **buf** — an array containing one element for each of the **nservers** servers. This array is stored in a system buffer, and it is the address of this system buffer that is returned in **buf**. After the program is finished using this information, this buffer should be returned by calling **bpfs_return_buffer**.

  Each item in this array is a structure of type **bpfs_server_status_t**. The only field in this structure that contains useful information is the field **server_location**, which is of type **bpfs_location_t**. This is an opaque structure containing the internal representation of the server's implementation-defined network address.

2.8 Closing a File

Once all I/O operations have been finished on an open file, the programmer closes the file by calling the pair:

```
extern char *bpfs_start_close( PFILE *pfptr );

extern char *bpfs_wait_for_close( PFILE *pfptr );
```

Once the function **bpfs_start_close** has been called on a file pointed to by **pfptr**, no other operations are allowed on that file pointer except the call to **bpfs_wait_for_close** itself, and once that has been called, nothing else can be done with that file pointer.

2.9 Buffered I/O

Buffered I/O is a style of reading and writing in which the user manipulates system buffers in each I/O operation. When reading, the read operations return a pointer to a system buffer that contains the newly read data, and the user uses that data out of the buffer. This buffer must then be returned to the system when the program is finished using the data it contains. When writing, the user must obtain a buffer from the system, put data into it, and then give a pointer to that buffer to the write operations in order to have it written to the file.

2.9.1 Buffered Reading from a File

Once a parallel file has been opened for reading in BPFS_READ_ONLY or BPFS_READ_WRITE modes, the following operations can be used to read data from the file's disk storage area into system buffers:

```
extern char *bpfs_start_buffer_read( PFILE *pfptr,
        unsigned int start_block,
        unsigned int total_blocks );

extern char *bpfs_wait_for_buffer_read( PFILE *pfptr,
        unsigned int blockno,
        char **buf,
        unsigned int *nbytes );
```

The **bpfs_start_buffer_read** starts reading from the file opened on **pfptr** a sequence of **total_blocks** blocks of data starting with block number **start_block**. If **total_blocks** is zero, one block of data will be read.

For each of the **total_blocks** blocks requested in a call to the function **bpfs_start_buffer_read**, the program must perform a call to the function **bpfs_wait_for_buffer_read** specifying as input parameters the file pointer **pfptr** and **blockno**, the number of the next block the program wishes to process (blocks do not have to be processed in sequence). Each call to **bpfs_wait_for_buffer_read** will wait until the data from the indicated block is available in a system buffer. It then returns a pointer to that buffer in the

output parameter **buf**, and the number of bytes of data in that buffer in the output parameter **nbytes**. If **nbytes** is 0, then the block is at or beyond the end of the file, and no more blocks in this sequence will be received.

A buffered read cannot be made to any file on which block streaming has previously been turned on. If buffered streaming has been turned on, then only the **bpfs_wait_for_buffer_read** function needs to be called, as explained below in the section on buffer streaming.

One way these functions could be used in a segment of code is as follows:

```
message = bpfs_start_buffer_read(pfptr, first, number);
if( message == NULL )
    {/* read started correctly */
    for( i = first;  i < first+number;  i++ )
        {
        message = bpfs_wait_for_buffer_read(pfptr, i, &buf,
                                                &nbytes);
        if( message == NULL )
            {/* nbytes of data in buf */
            if( nbytes == 0 )
                {/* hit end of file, buffer is empty */
                bpfs_return_buffer(buf);
                break;
                }
            ... process the data ...
            bpfs_return_buffer(buf);
            }
        else
            {/* detected an error on the read */
            bpfs_perror("reading", message);
            break;
            }
        }
    }
...
```

2.9.2 Buffered Writing to a File

Once a parallel file has been opened for writing in BPFS_WRITE_ONLY or BPFS_READ_WRITE modes, the following operations can be used to write data from a system buffer to the disk storage area of the file:

```
extern char *bpfs_start_buffer_write( PFILE *pfptr,
        unsigned int blockno,
        char *buf,
        unsigned int nbytes );

extern char *bpfs_wait_for_buffer_write( PFILE *pfptr,
        unsigned int blockno,
        unsigned int *nbytes );
```

The **bpfs_start_buffer_write** starts writing **nbytes** of data from the buffer pointed to by **buf** to block number **blockno** in the file opened on **pfptr**. The value in **nbytes** must not be greater that BLOCKSIZE.

If the file was opened with the **open_no_reply_on_write** attribute set to 0, this indicates that the programmer wants to receive notification after the block has been written onto the server disk. To do this the program must call the function **bpfs_wait_for_buffer_write** specifying as input parameters the file pointer **pfptr** and **blockno**, the number of the block that was written previously in the call to **bpfs_start_buffer_write**. The function **bpfs_wait_for_buffer_write** will wait until the server of this block has replied with the number of bytes of data actually written to disk, and will then return this number in its output parameter **nbytes**. Normally, of course, this will have the same value as the input parameter **nbytes** to the function **bpfs_start_buffer_write**. However, if an error occurred on the write to disk, the error message will be returned as the value of the function **bpfs_wait_for_buffer_write**.

If the file was opened with the **open_no_reply_on_write** attribute set to 1 (which is the normal default), then the programmer does not want to receive notification after the block has been written onto the server disk, and **bpfs_wait_for_buffer_write** need not be called (if it is called, it will immediately return with a NULL pointer and a value of BLOCKSIZE stored in **nbytes**). In this case, notification of any error that might have occurred on the write to disk is lost.

Whether a reply is expected or not, the buffer pointed to by **buf** is always freed by **bpfs_start_buffer_write**, and should no longer be referenced for any purpose by the program after the call to this function.

The following segment of code shows one way these functions could be used when the programmer wishes to receive a reply from the server after writing every block:

```
message = bpfs_get_buffer(&buf);
if( message == NULL )
    {
    ... fill the buffer with nbytes of data ...
    message=bpfs_start_buffer_write(pfptr,blockno,buf,nbytes);
    if( message == NULL )
        {/* write started ok, wait for server to reply */
        message = bpfs_wait_for_buffer_write(pfptr, blockno,
                                                &nactual);
        if( message == NULL )
            {/* correctly wrote nactual bytes to disk */
            }
        else
            {/* data not written to disk correctly */
            bpfs_perror("wait for write", message);
            }
        }
    else
        {/* detected an error when starting the write */
        bpfs_perror("start write", message);
        }
    }
...
```

This code becomes much simpler if the programmer does not wish to receive a verification from the server after writing every block (which is the default case):

```
message = bpfs_get_buffer(&buf);
if( message = NULL )
    {
    ... fill the buffer with nbytes of data ...
    message=bpfs_start_buffer_write(pfptr,blockno,buf,nbytes);
    if( message != NULL )
        {/* detected an error when starting the write */
        bpfs_perror("start write", message);
        }
    }
...
```

2.9.3 Buffered Data Streaming from a File

Once a parallel file has been opened for reading in BPFS_READ_ONLY mode, the following operations can be used to turn on and off data streams from the file's disk storage area into system buffers. Once a data stream is started, the servers in the parallel file system send blocks of data to the client without waiting for the client to send explicit requests for the blocks. This "data push" mode of operation cuts down on network traffic by eliminating the requests, and improves response time by effectively "prefetching" all the blocks in the stream.

```
extern char *bpfs_start_buffer_stream_on( PFILE *pfptr,
        unsigned int start_block,
        unsigned int stride,
        unsigned int total_blocks,
        unsigned int rate_blocks,
        unsigned int rate_seconds );

extern char *bpfs_wait_for_buffer_stream_on( PFILE *pfptr );

extern char *bpfs_start_stream_off( PFILE *pfptr );

extern char *bpfs_wait_for_stream_off( PFILE *pfptr );
```

The **bpfs_start_buffer_stream_on** starts streaming data from the file opened on **pfptr**. This stream is a sequence of **total_blocks** blocks of data starting with block number **start_block** and including only every **stride**th block in the file. These blocks will be sent at the rate of **rate_blocks** blocks every **rate_seconds** seconds.

If **total_blocks** is zero, then the sequence continues to the end of the file. If either **rate_blocks** or **rate_seconds** is zero, blocks will be sent "as fast as possible". However, if **rate_blocks** is non-zero and **rate_seconds** is zero, then **rate_blocks** is the maximum number of buffers that can be queued up on the client side waiting to be read. If this number is exceeded, buffers are "backed up" in the network layer and the servers will be forced to delay sending any further buffers until the backlog is cleared. If the application program does not specify an explicit maximum number, the system will utilize a default value.

Note that the rates are given in blocks per second. To specify bytes per second, set **rate_blocks** to the rate in bytes per second and **rate_seconds** to BLOCKSIZE.

For each of the **total_blocks** blocks requested in the call to function **bpfs_start_buffer_stream_on**, the program must perform a call to the function **bpfs_wait_for_buffer_read** specifying as input parameters the file pointer **pfptr** and **blockno**, the number of the next block in the stream that the program wishes to process. Blocks do not have to be processed in the same sequence as they are pushed by the server, but the program can deadlock itself if it gets so far out of sequence that the number of backed up buffers exceeds the flow control maximum. **bpfs_wait_for_buffer_read** will wait until the data from the indicated block is available in a system buffer, and will then return a pointer to that buffer in the output parameter **buf**, and the number of bytes of data in that buffer in the output parameter **nbytes**. If a value of 0 is returned in **nbytes**, then the block is at or beyond the end of the file, and the stream is automatically stopped. The stream also stops if an error is detected.

Note that **bpfs_start_buffer_read** need not be called if buffer streaming has been turned on, but if it is called, it simply returns (successfully) without doing anything. Note also that the program must take care to wait only for buffers that are generated in the sequence given by the **start_block** and **stride** values. Finally, note that it may not be possible to start a stream for all possible combinations of mapping functions, number of servers, thickness values, and strides.

The following shows one way these functions could be used in a segment of code to read a stream containing all blocks (**stride** = 1) in a file starting with block number **first**:

```
message = bpfs_start_buffer_stream_on(pfptr,first,1,0,0,0);
if( message == NULL )
    {
    message = bpfs_wait_for_buffer_stream_on(pfptr);
    if( message != NULL )
        pfs_perror("wait for stream on", message);
    else
        {/* stream started correctly */
        for( i = first;  ; i++  )
            {
            message = bpfs_wait_for_buffer_read(pfptr, i,
                                        &buf, &nbytes);
            if( message == NULL )
                {/* nbytes of data in buf */
                if( nbytes == 0 )
                    {/* hit end of file, stream stops */
                    bpfs_return_buffer(buf);
                    break;
                    }
                ... process the data ...
                bpfs_return_buffer(buf);
                }
            else
                {/* detected an error on the read */
                bpfs_perror("reading", message);
                break;
                }
            }
        message = bpfs_start_stream_off(pfptr);
        if( message == NULL )
            message = bpfs_wait_for_stream_off(pfptr);
        }
    }
...
```

2.10 Direct I/O

Direct I/O is a style of reading and writing in which the user specifies areas of memory into which data blocks are read directly or out of which data blocks are written directly. This is the "normal" method of performing I/O in higher-level languages, except that it is performed here in units of blocks

rather than bytes. There are no visible system buffers involved in this style of I/O (except the buffers containing error messages that are returned by all API0 functions when an error is detected). Direct and buffered I/O can be mixed on the same open file, although both types of streams cannot be simultaneously active on the same open file.

2.10.1 Direct Reading from a File

Once a parallel file has been opened for reading in BPFS_READ_ONLY or BPFS_READ_WRITE modes, the following operations can be used to read data from the file's disk storage area directly into an area of user memory:

```
extern char *bpfs_start_block_read( PFILE *pfptr,
        unsigned int start_block,
        unsigned int total_blocks,
        char *storage );

extern char *bpfs_wait_for_block_read( PFILE *pfptr,
        unsigned int start_block,
        unsigned int *nbytes );
```

The function **bpfs_start_block_read** starts reading from the open file **pfptr** a sequence of **total_blocks** blocks of data starting with block number **start_block**. If **total_blocks** is zero, one block of data will be read. The data contained in these blocks will be stored into the area of user memory pointed to by **storage**. This must be big enough to hold a total of **total_blocks** $\times$ BLOCKSIZE bytes, and must be aligned on a 32-bit word boundary.

Once the function **bpfs_start_block_read** has returned successfully, the data in the area of user memory pointed to by **storage** is undefined until after the program has called the function **bpfs_wait_for_block_read** specifying as input parameters the same file pointer **pfptr** and **start_block**. The call to **bpfs_wait_for_block_read** will wait until the data from all **total_blocks** blocks has been read into **storage**, and will then return the total number of bytes of data in all those blocks in the output parameter **nbytes**. If **nbytes** is 0, then the blocks were at or beyond the end of the file and nothing has been stored in **storage**. If **nbytes** is less than **total_blocks** $\times$ BLOCKSIZE, then this request hit the end of the file and **storage** contains the rest of the file starting at **start_block**.

Although several calls to start a block read can be made before any calls to wait for them, the range of blocks specified in each outstanding block read

or write must not overlap those given in any other outstanding block read or write. Similarily, a block read cannot be performed if a buffer stream has been started on the same open file. (There is no problem if a block stream has been started.)

One way these functions could be used in a segment of code is as follows:

```
message = bpfs_start_block_read(pfptr, first, number, storage);
if( message == NULL )
    {
    message = bpfs_wait_for_block_read(pfptr, first, &nbytes);
    if( message == NULL )
        {/* nbytes of data in storage */
        if( nbytes > 0 )
            {
            ... process the data ...
            }
        }
    else
        {/* detected an error on the read */
        bpfs_perror("reading", message);
        }
    }
...
```

2.10.2 Direct Writing to a File

Once a parallel file has been opened for writing in BPFS_WRITE_ONLY or BPFS_READ_WRITE modes, the following operations can be used to write data directly from an area of user memory to the file's disk storage area:

```
extern char *bpfs_start_block_write( PFILE *pfptr,
        unsigned int start_block,
        unsigned int total_blocks,
        const char *storage );

extern char *bpfs_start_partial_block_write( PFILE *pfptr,
        unsigned int start_block,
        unsigned int nbytes,
        const char *storage );
```

```
extern char *bpfs_wait_for_block_write( PFILE *pfptr,
        unsigned int start_block,
        unsigned int *nbytes );
```

The function **bpfs_start_block_write** starts to write **total_blocks** $\times$ BLOCKSIZE bytes of data from the user storage area pointed to by **storage** (which must be aligned on a 32-bit word boundary) to block number **start_block** in the file opened on **pfptr**. If **total_blocks** is zero, one block of data will be written.

The function **bpfs_start_partial_block_write** starts to write **nbytes** bytes of data (which must be no greater than BLOCKSIZE) from the user storage area pointed to by **storage** (which must be aligned on a 32-bit word boundary) to block number **start_block** in the file opened on **pfptr**. This function should be used only to write a partial block at the end of the file — if a partial block is written in the middle of the file, the remaining bytes in the block will be undefined when they are read back.

The programmer should not modify any of the data pointed to by **storage** until after calling the function **bpfs_wait_for_block_write** specifying as input parameters the same file pointer **pfptr** and **start_block** as were specified previously in a call to the function **bpfs_start_block_write** or the function **bpfs_start_partial_block_write**. The call to the function **bpfs_wait_for_block_write** will wait until all the blocks have been sent to the server and it is safe for the user to again modify the data pointed to by **storage**. The number of bytes of data actually sent to the servers will be returned in the output parameter **nbytes**.

Although several calls to start a block write can be made before any calls to wait for them, the range of blocks specified in each outstanding block write or read must not overlap those given in any other outstanding block write or read.

Note that the **open_no_reply_on_write** attribute of the open file is ignored on direct writes, since these write operations always complete only after all blocks have been sent to the server and it is again safe for the user to modify the data pointed to by **storage**.

The following segment of code shows one way these functions could be used to write **n** blocks of data that a user has generated in an area of his memory pointed to by **storage**. These blocks are to be written onto disk starting at block number **first**.

```
... fill the storage area with n blocks of data ...
message = bpfs_start_block_write(pfptr, first, n, storage);
if( message == NULL )
    {/* wait for data to be sent before touching storage */
    message = bpfs_wait_for_block_write(pfptr, first,
                                                &nactual);
    if( message == NULL )
        {/* actually sent nactual bytes to disk */
        }
    else
        {/* data not sent to disk correctly */
        bpfs_perror("wait for write", message);
        }
    }
else
    {/* detected an error when starting the write */
    bpfs_perror("start write", message);
    }
...
```

2.10.3 Direct Data Streaming from a File

Once a parallel file has been opened for reading in BPFS_READ_ONLY mode, the following operations can be used to turn on and off data streams from the file's disk storage area into areas of user memory. Once a data stream is started, the servers in the parallel file system send blocks of data to the client without waiting for the client to send explicit requests for the blocks.

```
extern char *bpfs_start_block_stream_on( PFILE *pfptr,
        unsigned int start_block,
        unsigned int stride,
        unsigned int total_blocks,
        unsigned int rate_blocks,
        unsigned int rate_seconds );

extern char *bpfs_wait_for_block_stream_on( PFILE *pfptr );

extern char *bpfs_start_stream_off( PFILE *pfptr );

extern char *bpfs_wait_for_stream_off( PFILE *pfptr );
```

The **bpfs_start_block_stream_on** starts streaming data from the file opened on **pfptr**. This stream is a sequence of **total_blocks** blocks of data starting with block number **start_block** and including only every **stride**th block in the file. These blocks will be sent at the rate of **rate_blocks** blocks every **rate_seconds** seconds.

If **total_blocks** is zero, then the sequence continues to the end of the file. If either **rate_blocks** or **rate_seconds** is zero, blocks will be sent "as fast as possible". However, if **rate_blocks** is non-zero and **rate_seconds** is zero, then **rate_blocks** is the maximum number of buffers that can be queued up on the client side waiting to be read. If this number would be exceeded, buffers are "backed up" in the network layer and the servers will be forced to delay sending any further buffers until the backlog is cleared.

Note that the rates are given in blocks per second. To specify bytes per second, set **rate_blocks** to the rate in bytes per second and **rate_seconds** to BLOCKSIZE.

Once a block stream has been turned on correctly, as determined by a successful return from **bpfs_wait_for_block_stream_on**, the program actually reads the data in the same way as it would read non-streamed data: by using the **bpfs_start_block_read** and **bpfs_wait_for_block_read** functions in exactly the same manner as described previously. However, care must be taken to ensure that these read functions only attempt to read blocks which will actually be generated by the stream. The main consequence of this requirement is that if the stream is turned on with a **stride** value greater than one, then the **total_blocks** parameter to **bpfs_start_block_read** must be exactly one, since if it were more than one it would attempt to read that number of consecutive blocks, and a **stride** value greater than one does not generate consecutive blocks. Looking at this from the other way around, if a call to **bpfs_start_block_read** is going to specify a value for its **total_blocks** parameter that is greater than one, then the stream must be turned on with a **stride** value of exactly one, in order to generate consecutive blocks.

The following shows one way these functions could be used in a segment of code to read a stream containing all blocks (**stride** = 1) in a file starting with block number **first**:

```
message = bpfs_start_block_stream_on(pfptr,first,1,0,0,0);
if( message == NULL )
    {
    message = bpfs_wait_for_block_stream_on(pfptr);
    if( message != NULL )
        pfs_perror("wait for stream on", message);
    else
        {/* stream started correctly */
        for( i = first;  ; i++  )
            {
            message = bpfs_start_block_read(pfptr,i,1,storage);
            if( message == NULL )
                {/* block read started correctly */
                message = bpfs_wait_for_block_read(pfptr, i,
                                                   &nbytes);
                if( message == NULL )
                    {/* nbytes of data in storage */
                    if( nbytes == 0 )
                        {/* hit end of file, stream stops */
                        break;
                        }
                    ... process the data ...
                    }
                else
                    {/* got an error reading, stream stops */
                    bpfs_perror("reading", message);
                    break;
                    }
                }
            }
        message = bpfs_start__stream_off(pfptr);
        if( message == NULL )
            message = bpfs_wait_for__stream_off(pfptr);
        }
    }
...
```

2.11 Synchronizing Data Written to a File

Successfully completed write operations in API0, whether buffered or direct, deliver data to the file servers. They do not guarantee that the data has been

written to the disk. Indeed, if the file servers are caching, it is possible that the server has not even attempted to write it to the disk. However, even if the server has done so, the host operating system may hold it in its cache, so that the data is still not on permanent disk storage.

In order to guarantee that data written to a parallel file has been flushed from all caches and forced onto permanent disk storage, the API0 programmer can use the following functions:

```
extern char *bpfs_start_sync( PFILE *pfptr );

extern char *bpfs_wait_for_sync( PFILE *pfptr );
```

The input parameter **pfptr** must be a file that has been successfully opened in BPFS_WRITE_ONLY or BPFS_READ_WRITE mode. The function **bpfs_start_sync** will send a message to all servers of the open file forcing them to empty their own caches for that file and to force their host operating systems to do likewise. When the function **bpfs_wait_for_sync** returns successfully, all data written to the file up to the time of the call to **bpfs_start_sync** is guaranteed to be on the server disks. No guarantee is made for any data written between the calls to these two functions, nor to data written after the call to **bpfs_wait_for_sync**.

This is an expensive operation, so it should be used sparingly by a programmer. The logical place to use it is just before closing the file pointed to by **pfptr**, since there is no automatic synchronization when a file is closed.

2.12 Server-side File Caching

2.12.1 Obtaining the Status of a Server Cache

```
extern char *bpfs_start_cache_status( PFILE *pfptr );

extern char *bpfs_wait_for_cache_status( PFILE *pfptr,
        unsigned int *nservers,
        unsigned int **buf );
```

This pair of operations allows the programmer to obtain the current status of all the servers serving the open file indicated by **pfptr**. The number of servers is returned in the output parameter **nservers**, and in the output parameter **buf** is returned a pointer to a system buffer containing an array containing six unsigned integers for each of the servers. These six values have the following meaning:

1. The on/off status of caching for this file on this server: 0 means OFF, 1 means ON. If this value is 0, all 5 remaining items will also be 0.

2. The cache replacement policy for this file on this server: 1 means FIFO, 2 means LRU, 3 means RANDOM. Higher values are possible if the system has defined additional policies.

3. The cache write-through policy for this file on this server: 1 means every write goes immediately to disk, 2 means blocks are written to disk only on removal from the cache.

4. The number of bins in the hash table for this file on this server.

5. The maximum number of buffers that can be kept in the cache at any one time for this file on this server.

6. The current number of buffers in the cache at this time for this file on this server.

Since the pointer returned in **buf** points to a system buffer, the programmer should return it to the free pool by calling **bpfs_return_buffer** when finished with it.

If either output parameter is NULL, the corresponding value is not returned.

2.12.2 Controlling Server Cache Operation

```
extern char *bpfs_start_cache_on( PFILE *pfptr,
        unsigned int *array );

extern char *bpfs_wait_for_cache_on( PFILE *pfptr,
        unsigned int *nservers,
        unsigned int **buf );

extern char *bpfs_start_cache_off( PFILE *pfptr );

extern char *bpfs_wait_for_cache_off( PFILE *pfptr );
```

These functions allow the programmer to turn caching on and off for an open file indicated by the input parameter **pfptr**.

When turning caching on, the user supplies as the input parameter **array** an array of six unsigned integer values having the meaning described above. A copy of this array will be sent to each of the servers serving the open file

pointed to by **pfptr**. The value in the first element of the array (the on/off value) is ignored. Any other element containing a zero value is interpreted to mean "use the system default" if caching was off prior to this operation to turn it on, and "leave unchanged" if caching was already on at the time of this operation to turn it on. An **array** parameter of NULL is equivalent to an array in which all six elements have zero values.

The symbol BPFS_CACHE_ON_INITIALIZER has been defined in the header file "api0.h" as a convenience to the programmer for use in initializing a newly declared **array** structure with default values (all zeros). This symbol could be used in a declaration of the array **xyz** as follows:

```
unsigned int  xyz[] = BPFS_CACHE_ON_INITIALIZER;
```

If this structure is not modified after initialization and it is passed as the **array** parameter to **bpfs_start_cache_on**, then the cache settings will have the same default values (all zeroes) as when a NULL pointer is passed as the **array** parameter to **bpfs_start_cache_on**.

The output parameters **nservers** and **buf** have exactly the same meaning as the output parameters for the cache status operations defined above, and they will reflect the new values after this start cache operation has taken effect.

The pointer returned in **buf** points to a system buffer, so the programmer should return it to the free pool by calling **bpfs_return_buffer** when finished with it.

If either output parameter is NULL, the corresponding value is not returned.

2.13 Managing Parallel Files

Because a parallel file consists of separate pieces of information that may reside on many different nodes in the network, it is difficult to manage such a file as a single entity without tools designed for that purpose. This section describes a set of these tools.

As might be expected, all of the functions for managing parallel files come in "start_ – wait_for_" pairs. The "start_" function always takes as input a parameter called **file_name**, which points to a null-terminated C string containing the name of the file to be managed. It always returns in an output parameter called **pfptr** a pointer to a file control block that can only be used as a required input parameter to the matching "wait_for_" function in the pair. It cannot be used for any other purpose, and is not in any sense a reference to an "open" file. It is used solely as a "handle" to convey

information about an on-going operation from the first function in the pair to the second. In the "wrapper" functions for these pairs, this **pfptr** parameter is omitted.

Each of these functions requires the file manager and each of the servers for the file to participate in its operation. Except for the "erase" function pair, an error detected by the manager or any of the servers causes the entire function to report an error and to take no action on the manager or on any of the servers. This guarantees that the parallel file will always be maintained in a consistent state.

2.13.1 Obtaining Information about Parallel Files

The following pair of functions is used to obtain status information about a parallel file whose name is given as the input parameter **file_name** to the **bpfs_start_status** function:

```
extern char *bpfs_start_status( const char *file_name,
        PFILE **pfptr );

extern char *bpfs_wait_for_status( PFILE *pfptr,
        unsigned int *mapping,
        unsigned int *thickness,
        unsigned int *nservers,
        bpfs_server_status_t **buf,
        unsigned int *total_blocks,
        unsigned int *extra_bytes );
```

Except for **pfptr**, all the other parameters to **bpfs_wait_for_status** are output parameters. These are the addresses of locations into which this function will store the indicated results. If any particular return value is not wanted, the corresponding parameter should be NULL. The results stored in non-NULL parameters are as follows:

- **mapping** — the number of the striping function associated with the file.
- **thickness** — the striping thickness associated with the file.
- **nservers** — the number of servers across which the file is striped.
- **buf** — an array of items, one item for each of the **nservers** servers serving this file. This array is stored in a system buffer, and it is the address of this system buffer that is returned in **buf**. After the program

is finished using this information, this buffer should be returned by calling **bpfs_return_buffer**.

Each item in this array is a structure of type **bpfs_server_status_t**. The fields in this structure which contain useful information are:

- **server_location** — a structure of type **bpfs_location_t** containing the internal representation of this server's implementation-defined network address.
- **server_protection** — an unsigned short containing the access rights to this file on this server. For a UNIX system, these rights take the form of an octal number containing 3 digits, as explained on page 10.
- **server_full_blocks** — an unsigned integer containing the number of full data blocks allocated to this file on this server.
- **server_extra_bytes** — an unsigned integer that contains the number of extra bytes in any partial block at the end of this file on this server.

- **total_blocks** — the total number of full data blocks allocated to this file on the disks of all **nservers** servers. For a regular striping pattern, all of these blocks will actually contain data, but for an irregular pattern, such as pseudo-random, more blocks are allocated than actually contain data.
- **extra_bytes** — the total number of bytes in partial blocks allocated to this file on the disks of all **nservers** servers. If they exist, these blocks are at the logical end of the file on each server.

2.13.2 Deleting Parallel Files

```
extern char *bpfs_start_delete( const char *file_name,
        PFILE **pfptr );

extern char *bpfs_wait_for_delete( PFILE *pfptr );

extern char *bpfs_start_erase( const char *file_name,
        PFILE **pfptr );

extern char *bpfs_wait_for_erase( PFILE *pfptr );
```

The difference between these two pairs of functions to delete a file is that the "delete" pair succeeds only if the parallel file is correctly constructed and would be accessible (i.e., all the servers are currently up and running), whereas the "erase" pair succeeds in more lenient circumstances. If any error is detected by the "delete", such as missing metadata or an undeleteable data file on a server, nothing is deleted and an error is returned from the **bpfs_wait_for_delete**.

The "erase" pair always erases whatever parts of a parallel file it can find, even when it returns an error to indicate that it did not totally succeed. For example, if one of the servers is inaccessible, the metadata and the data on all the remaining servers are nonetheless deleted, even though an error about the missing server will be returned.. The "erase" pair is useful for cleaning up when bugs or system crashes lead to incorrectly contructed parallel files.

2.13.3 Renaming Parallel Files

```
extern char *bpfs_start_move( const char *file_name,
        const char *new_file_name,
        PFILE **pfptr );

extern char *bpfs_wait_for_move( PFILE *pfptr );
```

This pair of functions changes the name of an existing parallel file from the name pointed to by the input parameter **file_name** (which must exist) to the name pointed to by the input parameter **new_file_name** (which must not exist). (In UNIX parlance this is called "moving a file".)

2.13.4 Linking Parallel Files

```
extern char *bpfs_start_link( const char *file_name,
        const char *new_file_name,
        PFILE **pfptr );

extern char *bpfs_wait_for_link( PFILE *pfptr );
```

This pair of functions gives an additional name or "alias" to an existing parallel file. (In UNIX parlance this is called "creating a link to a file".) The existing parallel file is named by the input parameter **file_name** and the additional name is given by the input parameter **new_file_name**. This additional name must not exist prior to this operation.

2.14 Location Conversion Functions

Two functions are provided to convert between the implementation-defined internal and external representations of a network location.

```
extern char *bpfs_strtolocation( const char *buf,
        const char *extra,
        bpfs_location_t *location );

extern char *bpfs_locationtostr( bpfs_location_t *location,
        char **buf );
```

bpfs_strtolocation converts from the external to the internal representation. There are two input parameters, **buf** and **extra**, both of which are pointers to null-terminated C strings. The external representation of the location is a merger of the information in these two parameters, with parts omitted in the first string taken from the second. The output parameter **location** is the address of the opaque structure into which this function stores the internal representation of the merger.

bpfs_locationtostr converts from the internal to the external representation. The input parameter, **location**, is a pointer to the opaque structure containing the internal representation. This function converts this to an external representation in the form of a null-terminated C string which it stores in a system buffer. The address of this system buffer is returned in the output parameter **buf**. After the program is finished using this string, this buffer should be returned by calling **bpfs_return_buffer**.

3 Standard C I/O Library Interface to BPFS

The Standard C I/O Library Interface to BPFS was created in order to make it easy to access BPFS from high-level applications which can be parallel or non-parallel programs. For example, this interface makes it possible for sequential programs to benefit from the higher throughput and greater storage capacity of a parallel file system, as well as some of the unique features of BPFS such as its data-streaming facility. An example application might be a video-on-demand application. This interface also makes it possible to create tools to convert between normal files and parallel files.

This interface, called "CLI" for "C Library Interface", exactly mimics a subset of the ANSI Standard [1] I/O interface, and each of the prototypes is given with a reference to its corresponding function in that standard. The following conventions were adopted to derive CLI from the standard interface:

- The name of a function in CLI is the same as the name of the corresponding function in the standard interface prefaced with the characters "p_". For example, "p_fopen", "p_fclose", etc.
- All references to the standard type "FILE" are replaced with references to the new type "P_FILE" in CLI functions.
- All references to the standard type "fpos_t" are replaced with references to the new type "p_fpos_t" in CLI functions.
- All CLI function prototypes and type declarations are defined in the header file "p_stdio.h", which must be included by every C source program utilizing this interface.
- CLI functions return the value EOF whenever the standard specifies a "non-zero" function return value. The standard requires EOF to be defined as a "negative integral" value in the header file "¡stdio.h¿".

The "p_" and "P_" prefixes are obviously intended to convey the notion of "parallel". The semantics of these functions is identical to the semantics of the corresponding functions in the standard. The only difference is that standard functions apply to files stored sequentially on the host file system, whereas "p_" functions apply to parallel files stored in parallel on BPFS.

3.1 Opening and Closing Parallel Files

```
/*    ANSI Standard C function 4.9.5.3, page 130    */
extern P_FILE *p_fopen( const char *filename,
      const char *mode );

/*    ANSI Standard C function 4.9.5.4, page 131    */
extern P_FILE *p_freopen( const char *filename,
      const char *mode, P_FILE *stream );

/*    ANSI Standard C function 4.9.5.1, page 129    */
extern int p_fclose( P_FILE *stream );
```

The function **p_fopen** opens the parallel file whose name is given by the the null-terminated C string pointed to by **filename**. How it is opened is determined by the first few characters in the null-terminated C string pointed to by **mode**, as shown in the following table:

mode	how file is opened
r	shared read
rb	shared read
r+	shared read and write
rb+	shared read and write
r+b	shared read and write
rbs	shared read-only with streaming
rs	shared read-only with streaming
w	shared write
wb	shared write
w+	shared read and write
wb+	shared read and write
w+b	shared read and write
ws	exclusive write-only
wbs	exclusive write-only

As in the Standard C Library, when the **mode** starts with the letter "r", the file whose name is given by the **filename** parameter must already exit or the open will fail. When the **mode** starts with the letter "w", if the named file already exists it will be truncated, if it does not exist it will be created. The letter "b" in any mode is ignored.

The letter "s" is non-standard, and is used to gain access to some of the features of the underlying parallel file system. A parallel file opened in "rs"

or "rbs" mode utilizes the "data streaming" feature of BPFS in which the file server "pushes" the data to the client program as rapidly as possible. This means that the program must read it from beginning to end sequentially, without seeking to different positions — calls to **p_fseek** and **p_fsetpos** are not allowed. Use of streaming greatly improves efficiency since it causes all the blocks in the file to be prefetched without waiting for explicit read requests. The "rs" or "rbs" mode allows simultaneous access only from other opens of the same type, and will fail if this file is already open in any other mode.

A parallel file opened in "ws" or "wbs" mode utilizes the "exclusive write-only" feature of BPFS which allows no simultaneous access from any other open stream The **p_fopen** will fail if this file is already open in any mode by any process.

The value returned by **p_fopen** is NULL on an error, otherwise it is a pointer to an open file control block that must be used as an input parameter to most of the other functions in this interface.

The function **p_fclose** closes the open file pointed to by its input parameter **stream**. No other operations may follow a successful **p_fclose** on this open file. This function returns 0 if successful, EOF otherwise.

The function **p_freopen** first closes the open file pointed to by its input parameter **stream** and then opens a new file whose name is given by **filename** in the mode given by **mode**. If this open is successful, the value returned by **p_freopen** will be the value of **stream**, which now points to the open control block for the new file. Any error on closing the file pointed to by **stream** is ignored. Any error on opening the new file causes **p_freopen** to return a NULL.

3.2 Flushing a Parallel File

```
/*    ANSI Standard C function 4.9.5.2, page 130    */
extern int p_fflush( P_FILE *stream );
```

This function forces to disk any data previously written to the open stream pointed to by **stream**. Nothing happens if this is not an output stream, or if no data has been written to it since the previous call to **p_fflush** or **p_fopen** or **p_freopen**. This function returns 0 if successful, EOF otherwise.

3.3 Reading from a Parallel File

```
/*    ANSI Standard C function 4.9.7.1, page 142    */
extern int p_fgetc( P_FILE *stream );

/*    ANSI Standard C function 4.9.8.1, page 145    */
extern size_t p_fread( void *ptr, size_t size,
      size_t nmemb, P_FILE *stream );
```

These two functions allow a program to read data from a parallel file. The function **p_fgetc** returns as its value the next character in the open file pointed to by **stream** if there is one, or EOF if there are no more characters in the stream. **stream** must be opened for reading or reading and writing. The current position within the stream advances by one after every successful call to **p_fgetc**.

The function **p_fread** attempts to read **nmemb** values, each consisting of **size** bytes, into the area of user storage pointed to by **ptr**. It returns the number of values successfully read, which is normally equal to **nmemb** but will be less than that on detection of the end of file or an error. The current position within the stream advances by the total number of bytes read into user storage.

3.4 Writing to a Parallel File

```
/*    ANSI Standard C function 4.9.7.3, page 142    */
extern int p_fputc( int c, P_FILE *stream );

/*    ANSI Standard C function 4.9.8.2, page 146    */
extern size_t p_fwrite( const void *ptr, size_t size,
      size_t nmemb, P_FILE *stream );
```

These two functions allow a program to write data to a parallel file. The function **p_fputc** writes the value of **c** as the next character in the open file pointed to by **stream**, which must be opened for writing or reading and writing. This function returns the value of **c** if successful, or EOF if an error occurs. The current position within the stream advances by one after every successful call to **p_fputc**.

The function **p_fwrite** writes **nmemb** values, each consisting of **size** bytes, from the area of user storage pointed to by **ptr**. It returns the number of values successfully written, which is normally equal to **nmemb** but will be less than that on detection of an error. The current position within the stream advances by the total number of bytes written from user storage.

3.5 Positioning within a Parallel File

```
/*    ANSI Standard C function 4.9.9.1, page 146    */
extern int p_fgetpos( P_FILE *stream, p_fpos_t *pos );

/*    ANSI Standard C function 4.9.9.3, page 147    */
extern int p_fsetpos( P_FILE *stream, const p_fpos_t *pos );

/*    ANSI Standard C function 4.9.9.4, page 147    */
extern long int p_ftell( P_FILE *stream );

/*    ANSI Standard C function 4.9.9.2, page 146    */
extern int p_fseek( P_FILE *stream, long int offset,
      int whence );

/*    ANSI Standard C function 4.9.9.5, page 147    */
extern void p_rewind( P_FILE *stream );
```

These five functions allow the program to obtain or change the current position within an open stream.

The function **p_fgetpos** saves into its output parameter **pos** the current position within **stream**. This value is represented in an implementation-defined manner that is capable of storing all possible positions in any size file that can be stored by the parallel file system. The only use for this saved value is as a later input parameter to the **p_fsetpos** function on the same open stream. **p_fgetpos** returns zero if it is successful; if not, it returns EOF and stores an error code in **errno**.

The function **p_fsetpos** restores the current position within **stream** to that represented by the value in the **pos** parameter, which must be a value previously saved by a call to **p_fgetpos** on the same open stream. **p_fsetpos** returns zero if it is successful; if not, it returns EOF and stores an error code in **errno**. Note that **p_fsetpos** always fails on a file opened in "rs", "rbs", "ws" or "wbs" modes.

The function **p_ftell** returns as its value the current position within **stream** specified as the number of characters from the beginning of the file. This will always be a non-negative value. Note that on 32-bit machines, the largest possible value that can be returned is 2,147,483,647 (2 Gigabytes), which may be less than the size of a parallel file. If the current position is beyond this value, or if any other error occurs, **p_ftell** returns -1L and stores an error code in **errno**.

The function **p_fseek** changes the current position within **stream** according to the values of its two input parameters **offset** and **whence**, as

shown in the following table:

whence	offset
SEEK_SET	absolute position
SEEK_CUR	relative to current position
SEEK_END	relative to end of file

The new position as calculated according to this table can never be negative. The value returned by a previous call to **p_ftell** can be used as the value of **offset** with a value of SEEK_SET for **whence** in order to restore a stream to its position at the time of the **p_ftell**. **p_fseek** returns zero if it is successful, EOF if it is not. Note that **p_fseek** always fails on a file opened in "rs", "rbs", "ws" or "wbs" modes.

The function call "**p_rewind(stream)**" is completely equivalent to the call "**(void)p_fseek(stream, 0, SEEK_SET)**". It too always fails on a file opened in "rs", "rbs", "ws" or "wbs" modes.

3.6 Error Handling for Parallel File Operations

```
/*    ANSI Standard C function 4.9.10.1, page 148    */
extern void p_clearerr( P_FILE *stream );

/*    ANSI Standard C function 4.9.10.2, page 148    */
extern int p_feof( P_FILE *stream );

/*    ANSI Standard C function 4.9.10.3, page 148    */
extern int p_ferror( P_FILE *stream );
```

These three functions allow the program to interrogate and clear the end of file and error indicators that are set on an open stream by the other I/O functions. Whenever a function, such as **p_fwrite**, detects an error, it sets an "error indication" flag that is associated with the stream. The function **p_ferror** returns EOF if this flag is set on the stream pointed to by its input parameter **stream**, and zero if this flag is not set. Similarly, whenever a function, such as **p_fread**, detects the end of file, it sets an "end of file indication" flag that is associated with the stream. The function **p_feof** returns EOF if this flag is set on the stream pointed to by its input parameter **stream**, and zero if this flag is not set. Both of these flags are "sticky" and are only reset by calling certain functions. The sole purpose of the function **p_clearerr** is to reset these two flags. However, other functions may also reset one or both of these flags, as shown in the following table:

function	error set	error reset	eof set	eof reset
p_fopen	no	if ok	no	if ok
p_freopen	no	if ok	no	if ok
p_fclose	if bad	no	no	no
p_fflush	if bad	no	no	no
p_fgetc	if bad	no	if eof	no
p_fread	if bad	no	if eof	no
p_fputc	if bad	no	no	no
p_fwrite	if bad	no	no	no
p_fgetpos	no	no	no	no
p_fsetpos	if bad	no	no	if ok
p_ftell	no	no	no	no
p_fseek	if bad	no	no	if ok
p_rewind	if bad	if ok	no	if ok
p_clearerr	no	always	no	always
p_feof	no	no	no	no
p_ferror	no	no	no	no

3.7 Managing a Parallel File

```
/*    ANSI Standard C function 4.9.4.1, page 128    */
extern int p_remove( const char *filename );

/*    ANSI Standard C function 4.9.4.2, page 128    */
extern int p_rename( const char *old, const char *new );
```

These two functions give a program minimal ability to manage a parallel file. **p_remove** removes (i.e., erases) the parallel file named by its input parameter **filename**. **p_rename** changes the name of a parallel file from the name given by **old**, which must exist, to the name given by **new**, which must not exist. Both functions return zero if they are successful, EOF if they are not.

4 MPI-IO over BPFS

MPI, which stands for "Message Passing Interface", is a well-established standard that has recently been extended to a new standard called MPI-2 [5]. The most important of the extensions for our work is the addition of I/O facilities, which have been given the name MPI-IO [3], and are described in Chapter 10 of [5].

ROMIO [8] is a high-performance, portable implementation of MPI-IO that is freely available at http://www.mcs.anl.gov/home/thakur/romio. The key to ROMIO's portability is its implementation on top of ADIO [7], which stands for "Abstract-Device interface for parallel I/O". In order to port ROMIO to a new machine or file system, only the ADIO needs to be rewritten. The distribution of version 1.0.0 (October 1997) of ROMIO contains several implementations of ADIO, including one for a generic UNIX file system, one for SUN's NFS, one for Intel's PFS, and one for IBM's PIOFS. In order to port ROMIO to use BPFS, we simply wrote a new implementation of ADIO based on the generic UNIX version supplied in the ROMIO distribution. This implementation provides all the functionality of the generic UNIX version, except that the files are accessed using BPFS.

One of the features of ROMIO is that several different parallel file systems can be accessed simultaneously by a single running parallel program. In order to accomplish this, ROMIO requires that every file name passed to MPI_File_open must have as a prefix the name of the file system on which it is stored. The prefixes supported by the standard distribution are "ufs:", "nfs:", "pfs:", and "piofs:", one for each of the implementations of ADIO mentioned above. To this set we have added the prefix "bpfs:" to correspond to the new implementation of ADIO for BPFS.

When MPI_File_open is called to open a file, the third argument must have one of the three values shown in the following table, possibly OR'd with other values defined in the MPI-IO standard. The interpretation of these "modes" in BPFS is shown in the table. MPI_MODE_RDWR is the most general mode, and it is anticipated that most programs should use this mode, since the other two modes restrict the access to parallel files.

MPI-IO mode	Meaning in BPFS
MPI_MODE_RDONLY	shared read-only muliple users, all must be read-only
MPI_MODE_WRONLY	exclusive write-only only one user
MPI_MODE_RDWR	shared read-write multiple users, all must be read-write

Although ROMIO was originally written for and tested on the MPICH [4] implementation of MPI, we have only tested the implementation of ADIO for BPFS on the LAM [2] implementation of MPI, since that is the only version available to us at this time.

References

[1] *American National Standard for Information Systems — Programming Language — C*, volume ANSI X3.159-1989. American National Standards Institute, 1990.

[2] Ohio Supercomputer Center. LAM: Local area multicomputer. http://www.osc.edu/lam.html.

[3] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

[4] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. MPICH: A high-performance, portable implementation of the MPI message passing interface standard. http://www.mcs.anl.gov/mpi/mpich.

[5] MPI-2: Extensions to the message-passing interface. The MPI Forum, July 1997.

[6] Robert D. Russell. The architecture of BPFS: a basic parallel file system, version 1.0. Technical report, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Lyon, France, March 1998.

[7] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[8] Rajeev Thakur, Ewing Lusk, and William Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.

Appendix

A.1 Alphabetic List of API0 Function Prototypes

The following list contains the prototypes of all the API0 functions, exactly as they appear in the header file "api0.h".

```
extern unsigned int bpfs_blocksize( void );

extern char *bpfs_block_read( PFILE *pfptr,
        unsigned int start_block,
        unsigned int total_blocks,
        char *storage,
        unsigned int *nbytes );

extern char *bpfs_block_stream_on( PFILE *pfptr,
        unsigned int startblock,
        unsigned int stride,
        unsigned int totalblocks,
        unsigned int rateblocks,
        unsigned int rateseconds );

extern char *bpfs_block_write( PFILE *pfptr,
        unsigned int start_block,
        unsigned int total_blocks,
        const char *storage,
        unsigned int *nbytes );

extern char *bpfs_buffer_read( PFILE *pfptr,
        unsigned int blockno,
        char **buf,
        unsigned int *nbytes );

extern char *bpfs_buffer_stream_on( PFILE *pfptr,
        unsigned int startblock,
        unsigned int stride,
        unsigned int totalblocks,
        unsigned int rateblocks,
        unsigned int rateseconds );
```

```
extern char *bpfs_buffer_write( PFILE *pfptr,
        unsigned int blockno,
        char *buf,
        unsigned int *nbytes );

extern char *bpfs_cache_off( PFILE *pfptr );

extern char *bpfs_cache_on( PFILE *pfptr,
        unsigned int *nservers,
        unsigned int *newsettings,
        unsigned int **reply );

extern char *bpfs_cache_status( PFILE *pfptr,
        unsigned int *nservers,
        unsigned int **buf );

extern char *bpfs_close( PFILE *pfptr );

extern char *bpfs_delete( const char *file_name );

extern char *bpfs_endup( void );

extern char *bpfs_erase( const char *file_name );

extern char *bpfs_get_buffer( char **buf );

extern char *bpfs_link( const char *existing_file_name,
        const char *new_file_name );

extern char *bpfs_locationtostr(struct bpfs_location *location,
        char **buf );

extern char *bpfs_move( const char *existing_file_name,
        const char *new_file_name );

extern char *bpfs_open( const char *file_name,
        unsigned int mode,
        struct bpfs_open_attributes *attr,
        PFILE **pfptr );
```

```
extern char *bpfs_open_information( PFILE *pfptr,
        char **file_name,
        unsigned int *mode,
        unsigned int *mapping,
        unsigned int *thickness,
        unsigned int *nservers,
        struct bpfs_server_status **buf );

extern char *bpfs_partial_block_write( PFILE *pfptr,
        unsigned int start_block,
        unsigned int in_bytes,
        const char *storage,
        unsigned int *out_bytes );

extern void bpfs_perror( const char *info,
        char *buf );

extern char *bpfs_return_buffer( char *buf );

extern char *bpfs_setup( unsigned int dump_stuff );

extern char *bpfs_start_block_read( PFILE *pfptr,
        unsigned int start_block,
        unsigned int total_blocks,
        char *storage );

extern char *bpfs_start_block_stream_on( PFILE *pfptr,
        unsigned int startblock,
        unsigned int stride,
        unsigned int totalblocks,
        unsigned int rateblocks,
        unsigned int rateseconds );

extern char *bpfs_start_block_write( PFILE *pfptr,
        unsigned int start_block,
        unsigned int total_blocks,
        const char *storage );

extern char *bpfs_start_buffer_read( PFILE *pfptr,
        unsigned int start_block,
        unsigned int total_blocks );
```

```
extern char *bpfs_start_buffer_stream_on( PFILE *pfptr,
        unsigned int startblock,
        unsigned int stride,
        unsigned int totalblocks,
        unsigned int rateblocks,
        unsigned int rateseconds );

extern char *bpfs_start_buffer_write( PFILE *pfptr,
        unsigned int blockno,
        char *buf,
        unsigned int nbytes );

extern char *bpfs_start_cache_off( PFILE *pfptr );

extern char *bpfs_start_cache_on( PFILE *pfptr,
        unsigned int *buf );

extern char *bpfs_start_cache_status( PFILE *pfptr );

extern char *bpfs_start_close( PFILE *pfptr );

extern char *bpfs_start_delete( const char *file_name,
        PFILE **pfptr );

extern char *bpfs_start_erase( const char *file_name,
        PFILE **pfptr );

extern char *bpfs_start_link( const char *existing_file_name,
        const char *new_file_name,
        PFILE **pfptr );

extern char *bpfs_start_move( const char *existing_file_name,
        const char *new_file_name,
        PFILE **pfptr );

extern char *bpfs_start_open( const char *file_name,
        unsigned int mode,
        struct bpfs_open_attributes *attr,
        PFILE **pfptr );
```

```
extern char *bpfs_start_partial_block_write( PFILE *pfptr,
        unsigned int start_block,
        unsigned int nbytes,
        const char *storage );

extern char *bpfs_start_status( const char *file_name,
        PFILE **pfptr );

extern char *bpfs_start_stream_off( PFILE *pfptr );

extern char *bpfs_start_sync( PFILE *pfptr );

extern char *bpfs_status( const char *file_name,
        unsigned int *mapping,
        unsigned int *thickness,
        unsigned int *nservers,
        struct bpfs_server_status **buf,
        unsigned int *total_bytes,
        unsigned int *extra_bytes );

extern char *bpfs_stream_off( PFILE *pfptr );

extern char *bpfs_strtolocation( const char *buf,
        const char *extra,
        struct bpfs_location *location );

extern char *bpfs_sync( PFILE *pfptr );

extern char *bpfs_wait_for_block_read( PFILE *pfptr,
        unsigned int start_block,
        unsigned int *nbytes );

extern char *bpfs_wait_for_block_stream_on( PFILE *pfptr );

extern char *bpfs_wait_for_block_write( PFILE *pfptr,
        unsigned int start_block,
        unsigned int *nbytes );

extern char *bpfs_wait_for_buffer_read( PFILE *pfptr,
        unsigned int blockno,
        char **buf,
```

```
        unsigned int *nbytes );

extern char *bpfs_wait_for_buffer_stream_on( PFILE *pfptr );

extern char *bpfs_wait_for_buffer_write( PFILE *pfptr,
        unsigned int blockno,
        unsigned int *nbytes );

extern char *bpfs_wait_for_cache_off( PFILE *pfptr );

extern char *bpfs_wait_for_cache_on( PFILE *pfptr,
        unsigned int *nservers,
        unsigned int **buf );

extern char *bpfs_wait_for_cache_status( PFILE *pfptr,
        unsigned int *nservers,
        unsigned int **buf );

extern char *bpfs_wait_for_close( PFILE *pfptr );

extern char *bpfs_wait_for_delete( PFILE *pfptr );

extern char *bpfs_wait_for_erase( PFILE *pfptr );

extern char *bpfs_wait_for_link( PFILE *pfptr );

extern char *bpfs_wait_for_move( PFILE *pfptr );

extern char *bpfs_wait_for_open( PFILE *pfptr );

extern char *bpfs_wait_for_status( PFILE *pfptr,
        unsigned int *mapping,
        unsigned int *thickness,
        unsigned int *nservers,
        struct bpfs_server_status **buf,
        unsigned int *total_bytes,
        unsigned int *extra_bytes );

extern char *bpfs_wait_for_stream_off( PFILE *pfptr );

extern char *bpfs_wait_for_sync( PFILE *pfptr );
```

A.2 Alphabetic List of CLI Function Prototypes

The following list contains the prototypes of all the CLI functions, exactly as they appear in the header file "p_stdio.h". These function prototypes are analogous to those in the ANSI C standard I/O library as defined in the document ANSI X.3159-1989 [1].

```
/*  ANSI Standard C function 4.9.10.1, page 148  */
extern void p_clearerr( P_FILE *stream );

/*  ANSI Standard C function 4.9.5.1, page 129  */
extern int p_fclose( P_FILE *stream );

/*  ANSI Standard C function 4.9.10.2, page 148  */
extern int p_feof( P_FILE *stream );

/*  ANSI Standard C function 4.9.10.3, page 148  */
extern int p_ferror( P_FILE *stream );

/*  ANSI Standard C function 4.9.5.2, page 130  */
extern int p_fflush( P_FILE *stream );

/*  ANSI Standard C function 4.9.7.1, page 142  */
extern int p_fgetc( P_FILE *stream );

/*  ANSI Standard C function 4.9.9.1, page 146  */
extern int p_fgetpos( P_FILE *stream, p_fpos_t *pos );

/*  ANSI Standard C function 4.9.5.3, page 130  */
extern P_FILE *p_fopen( const char *filename,
        const char *mode );

/*  ANSI Standard C function 4.9.7.3, page 142  */
extern int p_fputc( int c, P_FILE *stream );

/*  ANSI Standard C function 4.9.8.1, page 145  */
extern size_t p_fread( void *ptr, size_t size, size_t nmemb,
        P_FILE *stream );

/*  ANSI Standard C function 4.9.5.4, page 131  */
extern P_FILE *p_freopen( const char *filename,
        const char *mode, P_FILE *stream );
```

```
/*  ANSI Standard C function 4.9.9.2, page 146  */
extern int p_fseek( P_FILE *stream, long int offset,
        int whence );

/*  ANSI Standard C function 4.9.9.3, page 147  */
extern int p_fsetpos( P_FILE *stream, const p_fpos_t *pos );

/*  ANSI Standard C function 4.9.9.4, page 147  */
extern long int p_ftell( P_FILE *stream );

/*  ANSI Standard C function 4.9.8.2, page 146  */
extern size_t p_fwrite( const void *ptr, size_t size,
        size_t nmemb, P_FILE *stream );

/*  ANSI Standard C function 4.9.4.1, page 128  */
extern int p_remove( const char *filename );

/*  ANSI Standard C function 4.9.4.2, page 128  */
extern int p_rename( const char *old, const char *new );

/*  ANSI Standard C function 4.9.9.5, page 147  */
extern void p_rewind( P_FILE *stream );
```