Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation
LRR–TUM
D-80290 München

Users' Guide and Reference Manual
Version V1_0

Stefan Lamberts
pfslib@informatik.tu-muenchen.de

November 30, 1995

Contents

1 Introduction

PFSLib offers parallel access to files for a parallel application running on a cluster of workstations. It is intended but not restricted to be used in message passing applications based on PVM [11], NXLib [9, 8], MPI [6, 12], and other.

PFSLib emulates the parallel file system PFS of Intel's Paragon supercomputer system. Please refer to Intel's Paragon manuals[1] [3, 2] for an introduction to parallel I/O and Intel's PFS file access modes. A more detailed description of PFSLib itself is given in [1, 4, 5, 7].

If you have any questions, please send an email to `pfslib@informatik.tu-muenchen.de`.

2 Copying

PFSLib consists of a LIBRARY, daemon PROGRAMS, and utility PROGRAMS.

PFSLib is free software; you can redistribute the LIBRARY and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. You can redistribute the daemon PROGRAMS and utility PROGRAMS and/or modify them under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

PFSLib is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License and GNU General Public License for more details.

You should have received a copy of the GNU Library General Public License and the GNU General Public License along with this library; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

3 Security and Authentication

PFSLib uses Sun RPC communication between application processes and PFSLib server processes. To insure that unauthorized users can not get access to files via PFSLib the `AUTH_UNIX` authentication of the RPC facility is used. See a description of the Sun RPC facility for more detail on RPC authentication [10].

4 Availability and Installation

4.1 Obtaining PFSLib

PFSLib is available via anonymous ftp. Table 1 shows the information which is necessary to ftp PFSLib. As the software and the Users' Guide are provided in compressed format, make sure that you have set the `binary` mode for the ftp transmission.

Server:	ftpbode.informatik.tu-muenchen.de
	131.159.0.156
Login:	`anonymous`
Password:	your email address
Directory:	PFSLib
Files:	`PFSLib_V1_0.tar.Z`, `PFSLib_V1_0_UG.ps.Z` or
	`PFSLib_V1_0.tar.gz`, `PFSLib_V1_0_UG.ps.gz`

Table 1: Availability of PFSLib via anonymous ftp

[1] `http://www.ccsf.caltech.edu/paragon/man.html`

4.2 Unpacking

After the successful transmission of the files you will have to unpack them by the following commands

```
# uncompress PFSLib_V1_0.tar.Z
# tar xvf PFSLib_V1_0.tar
```

or

```
# gunzip PFSLib_V1_0.tar.gz
# tar xvf PFSLib_V1_0.tar
```

This results in a directory called `PFSLib_V1_0` which has been created in the current working directory. It contains all source files required to install PFSLib.

4.3 Installation procedure

Once you have successfully unpacked the source code and `PFSLib_V1_0` has been created, go to that directory:

```
# cd PFSLib_V1_0
```

After that you must edit the `Makefile` and configure the appropriate architecture and the appropriate installation directory of the libraries and binaries. For that purpose, you just have to choose one of the `ARCH` macros. Change the value of the `INSTALLDIR` macro if necessary. An installation can either be done in one's home directory (e.g. /home/lamberts/`PFSLib_V1_0`_SUN4) or in the system area (e.g. /usr/local/`PFSLib_V1_0`_SUN4).

Depending on the compiler you want to use, you have to check the compiler macro in the architecture makefiles, e.g. `Makefile.SUN4`. The current setting uses gcc on all architectures. If you do not have gcc you can use the vendor's compiler with the appropriate settings.

After all files have been edited and the macros have been set correctly the final step of the installation is to compile and install the libraries and binaries. This is done by a simple `make`:

```
# make
```

During that the installation directory gets populated with several subdirectories and all necessary files to use PFSLib.

4.4 Currently supported platforms

Table 2 gives an overview over all supported platforms.

Vendor	Type	Operating system	ARCH macro
Sun	SPARC	SunOS 4.1.x	SUN4
HP	9000	HP UX 09.01	HPPA
IBM	RS 6000	AIX 3.2	RS6K
Sun	SPARC	Solaris 2	SUN4SOL2
	PCs	Linux	LINUX

Table 2: Supported platforms

PFSLib should compile and run on other Unix platforms, too. They are not listed in the supported platforms since it could not be tested.

5 Using PFSLib

5.1 User installation

A prerequisite to use PFSLib within an application for every user is the following:

- Add the installations `bin` directory to your path to have access to all PFSLib commands, e.g. PATH=$PATH:/home/lamberts/`PFSLib_V1_0`_SUN4/bin or for `sh` or `ksh`.
- Make sure that the C preprocessor is able to find PFSLib's include files by adding the appropriate -I option to the preprocessor or compiler options (e.g. -I/usr/local/`PFSLib_V1_0`_SUN4/include).
- Make sure the linker is able to find the library `libpfs.a` by adding the appropriate -L option to the compiler or linker options (e.g. -L/usr/local/`PFSLib_V1_0`_SUN4/lib).
- Link the library libpfs.a to the application by adding `-lpfs` to the compiler or linker options. On Solaris 2 machines, additionally `-lsocket` and `-lnsl` have to be linked to the application.
- Make sure a `pfsd` is running on the machine specified in `pfslib_init()` [2].

5.2 Example PFSLib application

In the following example we assume that a environment variable `PFSLIBPATH` is set to the directory where PFSLib is installed and `PFSLIBPATH/bin` is in the `PATH` environment variable.

The program in figure 1 shows how to initialize and use PFSLib in an application.

The source code of this example can be found in `examples/C/SIMPLE/simple.c`.

Before any PFSLib function can be called, the application must call `pfslib_init()` (line 17) in order to register itself at the `pfsd` server. The parameter of the call are the host name of the machine `pfsd` resides on (line 17), the number of clients which is the number of processes accessing the files (line 18), and the number of the current client (line 19). In the example only one process accesses the file. Hence, the number of clients is 1 and it's number is 0 (zero).

With `gopen()` the program opens a PFSLib file with I/O mode M_UNIX(line 22). Beware that the `pfsd` server actually opens the file. Hence, it might not reside in the file system of the local machine, but on the machine `pfsd` is running on.

The `cwrite()` call in line 27 write synchronously to the file. The `lseek()` call in line 30 sets the file position back to the beginning of the file. The `cread()` call in line 36 read the data back in synchronously. Finally, the `close()` call in line 38 closes the file. In order to tell the `pfsd` to close the file `close` is a C preprocessor macro. PFSLib distinguishes between regular files and PFSLib files and calls the standard `close()` call in case of a regular file.

First of all you have to compile the application using the appropriate compiler switches.

```
# cc -I$PFSLIBPATH/include -L$PFSLIBPATH/lib -o simple simple.c -lpfs
```

Then you have to start the `pfsd` server in the background or in another shell. Here the output will be redirected. The `pfsd` administers all file access and even carries out file I/O up to a certain threshold. By default the `pfsd` will start eight `iod` basic I/O daemons which carry out file I/O above the threshold. The `pfsd` takes the number of I/O daemons it shall start as command line argument. Here, `pfsd` will start one `iod`.

```
# pfsd 1 &
```

Now start the application program. The parameter of the program is the name of the machine `pfsd` is running on.

```
# ./simple `hostname`
```

The `pfsd` server can continue to run in order to serve other applications. Here, it will be terminated.

```
# pfsdexit `hostname`
```

[2] See manual pages for pfsd(1 PFSLib), iod(8 PFSLib), and pfslib_init(3 PFSLib).

```
#include <fcntl.h>
#include <sys/types.h>
#include <stdio.h>

#include <pfslib.h>              /* Include the pfslib header file */

#define FILE "file"
#define DATA "This is a test\n"

int main(int argc, char **argv)
{

  int fd;
  char rdata[100];

  /* Initialize PFSLib */
  pfslib_init(argv[1],             /* Host of pfsd */
              1,                   /* Number of clients */
              0);                  /* Number of this process */

  /* Open a file */
  fd = gopen(FILE,                 /* Name of file */
             O_CREAT | O_RDWR,     /* Unix open(2) open flags */
             M_UNIX,               /* I/O mode */
             0644);                /* Unix file access permissions */

  cwrite(fd, DATA, strlen(DATA)); /* Write data synchronously */

  /* Set file pointer to beginning of file */
  if (lseek(fd,0,SEEK_SET) != 0)
  {
    pfslib_perror("lseek()");
    exit(1);
   }

  cread(fd,rdata,strlen(DATA));   /* Read data synchronously */

  close(fd);                       /* Close the file */

  exit(0);
}
```

Figure 1: Initialization of a PFSLib application

5.3 NXLib Initialization

In order to use PFSLib in an NXLib application the following function has to be called by every process before issuing any other PFSLib call, where `PFSDHOST` is the hostname of the machine the `pfsd` program is running on.

```
pfslib_init(PFSDHOST,numnodes(),mynode());
```

A NXLib example application which performs maximum search in a number of equal-sized arrays can be found in the `examples/C/NXLib` subdirectory.

5.4 PVM Initialization

Figure 2 shows a possibility to initialize PFSLib in a PVM application.

```
...
 1      mytid = pvm_mytid();          /* enroll in pvm */
 2
 3      tids[0] = pvm_parent();
 4
 5      if(tids[0] < 0)               /* find out if I am parent or child */
 6      {                             /* I am the parent */
 7        tids[0] = mytid;
 8        mynum = 0;                  /* my number is 0 */
 9
10        /* start up nproc-1 copies of myself */
11        pvm_spawn(argv[0], &argv[1], PvmTaskDefault, "", nproc-1, &tids[1]);
12
13        /* multicast tids array to children */
14        pvm_initsend(PvmDataDefault);
15        pvm_pkint(tids, nproc, 1);
16        pvm_mcast(&tids[1], nproc-1, 0);
17      }
18      else
19      {                             /* I am a child */
20        pvm_recv(tids[0], 0);       /* receive tids array */
21        pvm_upkint(tids, nproc, 1);
22
23        /* Use the index in the tid array as rank */
24        for(i=1; i<nproc ; i++)
25          if(mytid == tids[i]) { mynum = i; break; }
26      }
27
28      pfslib_init(argv[1], nproc, mynum); /* Initialize PFSLib */
...
```

Figure 2: Initialization of a PFSLib in PVM

First of all, the PVM parent process starts a number of processes with `pvm_spawn()` (line 11) and sends the task ids to all created processes (line 14–16). Every process uses the index in the tids array as a unique process number in the range form 0 to `nproc`-1 (lines 7, 23–25). Then, `pfslib_init()` in line 28 initializes PFSLib and sets up the connection to the `pfsd`.

A PVM example application which performs a distributed integer matrix vector multiplication can be found in the `examples/C/PVM` subdirectory.

References

[1] Arndt Bode, Christian Röder, and Stefan Lamberts. PFSLib – Using Intel's parallel i/o interface on coupled workstations. In *1995 European Intel Supercomputer Users' Group Meeting*, September 1995. http://www.irisa.fr/EXTERNE/cmpi/EISUG95/proc.html.

[2] Intel Corporation, Supercomputer Systems Division, Beaverton, Oregon. *Paragon C System Calls Reference Manual*, 312 487-003 edition, June 1994. http://www.ccsf.caltech.edu/paragon/man.html.

[3] Intel Corporation, Supercomputer Systems Division, Beaverton, Oregon. *Paragon User's Guide*, 312 489-003 edition, June 1994. http://www.ccsf.caltech.edu/paragon/man.html.

[4] Stefan Lamberts, Thomas Ludwig, Christian Röder, and Arndt Bode. PFSLib — A file system for parallel programming environments. SFB-Bericht, SFB 0342, Technische Universität München, 80290 München, Germany, 1995. To be published.

[5] Thomas Ludwig and Stefan Lamberts. PFSLib — A parallel file system for workstation clusters. In Victor Malyshkin, editor, *3rd International.Conference on Parallel Computing Technologies*, number 964 in Lecture Notes in Computer Science, pages 246–251, St. Petersburg, September 1995. Springer-Verlag.

[6] Message Passing Interface Forum. MPI: A massage-passing interface standard, May 1994.

[7] Christian Röder, Stefan Lamberts, and Thomas Ludwig. PFSLib – An i/o interface for parallel programming environments on coupled workstations. In Jack Dongarra, Marc Gengler, Bernard Tourancheau, and Xavier Vigouroux, editors, *2nd European PVM Users' Group Meeting*, volume 5 of *Parallélisme, réseaux et répartition*, pages 59–64, Lyon, September 1995. Hermes, Paris.

[8] Georg Stellner, Arndt Bode, Stefan Lamberts, and Thomas Ludwig. NXLib — a parallel programming environment for workstation clusters. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, number 817 in Lecture Notes in Computer Science, pages 745–748, Athens, July 1994. Springer-Verlag.

[9] Georg Stellner, Matthias Schumanm, Stefan Lamberts, Thomas Ludwig, Arndt Bode, M. Kiehl, and R. Mehlhorn. Developing multicomputer applications on networks of workstations using NXLib. SFB-Bericht 342/17/93 A, SFB 342 Technische Universität München, 80290 München, Germany, December 1993.

[10] Sun Microsystems, Mountain View, CA, USA. *Network Programming Guide*, 800-3850-10 edition, March 1990.

[11] V. S. Sunderam, G.A Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences and trends. *Parallel Computing*, 20(4):531–545, April 1994.

[12] David W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, April 1994.

Reference Manual

pfsd (1 PFSLib) ——— PFSLib V1_0 ——— pfsd (1 PFSLib)

Name

pfsd — *PFSLib file access management server*

Synopsis

pfsd [number of I/O daemons]

Paramters

number of I/O daemons **pfsd** starts the specified number of basic I/O daemons. If no argument is specified **pfsd** starts 8 basic I/O daemons by default.

Description

pfsd is PFSLib's file access management server. **pfsd** coordinates file access to files using PFSLib. Additionally, it handles basic I/O operation with an amount of data up to a specified or default threshold. It starts basic I/O daemons **iod** which handle I/O operations above the treshold. Application processes connect to **pfsd** using Sun's RPC facility with AUTH_UNIX authentication.

Environment

The `PATH` environment variable is used to located the **iod** binary.

See also

iod(8 PFSLib)

Bugs

Currently, the user id of the application processes must be identical to the user id of **pfsd**. Hence, a single **pfsd** cannot serve more than one user.

Since PFSLib uses the portmapper facility two or more **pfsd** processes cannot run on a single machine.

pfsdexit (1 PFSLib) ____________ PFSLib V1_0 ____________ pfsdexit (1 PFSLib)

Name

pfsdexit — *tell the PFSLib file access management server to exit*

Synopsis

pfsdexit pfsd host

Paramters

pfsd host Host name the **pfsd** is running on.

Description

pfsdreset sends a RPC to a **pfsd** which causes the process to terminate all forked **iod** process and itself.

See also

open(3 PFSLib), gopen(3 PFSLib), pfsd(1 PFSLib), iod(8 PFSLib)

Bugs

The RPC call might return with an error if the **pfsd** exits before the **pfsdexit** program received the RPC reply.

pfsdreset (1 PFSLib) —— PFSLib V1_0 —— pfsdreset (1 PFSLib)

Name

pfsdreset — *reset the PFSLib file access management server*

Synopsis

pfsdreset pfsd host [filename . . .]

Paramters

pfsd host Host name the pfsd is running on.

filename Name of files to reset. The file name must be identical to the name used in the **open()** or **gopen()** call.

Description

pfsdreset resets the internal data structures of a **pfsd** for all files or specified files.

See also

open(3 PFSLib), gopen(3 PFSLib), pfsd(1 PFSLib)

pfsdstat (1 PFSLib) —— PFSLib V1_0 —— pfsdstat (1 PFSLib)

Name

pfsdstat — *Tell the PFSLib file access management server to print status information*

Synopsis

pfsdstat pfsd host [-f filename] [-i] [-v]

Paramters

pfsd host	Host name the pfsd is running on.
-f filename	Makes the **pfsd** and/or **iod** print status information for the specified file name only.
-i	Tells **iod** processes to print status information as well.
-v	Verbose flag. Makes **pfsd** print all its internal data structures no matter if they are currently used or not.

Description

pfsdstat sends a RPC request to a **pfsd** located an the machine called pfsd host which makes the **pfsd** print internal status information to `stdout`. By default the **pfsd** only prints information on data structures which are in use. This command is intended for debugging purposes.

See also

pfsd(1 PFSLib), iod(8 PFSLib)

Bugs

The status information will not be printed by the **pfsdstat** program but by the **pfsd** and **iod** programs.

close() (3 PFSLib) —— PFSLib V1_0 —— close() (3 PFSLib)

Name

close() — *Closes a single accessed or a shared file*

Synopsis

#include <pfslib.h>

`int` **pfslib_close** (`int` *FileDescriptor*)

#define **close pflsib_close**

Paramters

FileDescriptor File descriptor of a single accessed or shared file.

Description

The **pfslib_close()** function disconnects the process from the shared file or from a regular UNIX file. The **pfslib_close()** call in PFSLib behaves identically to the standard UNIX **close()** call for files.

For compatibility reasons to Intel's PFS, **close** is defined in `pfslib.h` as a C preprocessor macro to overlay the C-library call.

Return Values

On success **close** returns 0. On failure it returns -1 and sets *errno* to indicate the error.

Errors

If the **pfslib_close** function fails, *errno* may be set to one of the error code values set by the standard Unix **close()** function of the following value.

EPFSLAUTH PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.

EPFSLBADF PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.

EPFSLBADFH PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.

EPFSLMIXIO PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.

EPFSREMOTE PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

close(2), open(3 PFSLib), gopen(3 PFSLib), pfslib_perror(3 PFSLib)

cread() (3 PFSLib) ———— PFSLib V1_0 ———— cread() (3 PFSLib)

Name

cread() — *Reads from a shared file and blocks the calling process until the read completes. (Synchronous read)*

Synopsis

```
#include <pfslib.h>

void cread ( int FileDescriptor,
             char * Buffer,
             unsigned int NBytes )

long _cread ( int FileDescriptor,
              char * Buffer,
              unsigned int NBytes )

#include <uio.h>
#include <pfslib.h>

void creadv ( int FileDescriptor,
              struct iovec * iov,
              int iovCount )

long _creadv ( int FileDescriptor,
               struct iovec * iov,
               int iovCount )
```

Paramters

FileDescriptor File descriptor of a shared file.

Buffer Pointer to the buffer in which to store the data after it is read form the file.

NBytes Number of bytes to read from the file.

iov Pointer to an array of `iovec` structures that identifies the buffers into which the data read is placed. The iovec structure is defines in `sys/uio.h`.

iovCount Number of `iovec` structures pointed to by the *iov* parameter.

Description

Other than the return values and additional errors, the **cread()** and **creadv()** functions are identical to the **read()** and **readv()** functions, respectively.

These are synchronous calls. The calling process waits (blocks) until the read completes. Use the **iread()** and **ireadv()** functions to read from a file without blocking the calling process.

Reading past the end of a file causes an error. You can do one of the following to prevent end-of-file errors.

- Use the **iseof()** function to detect end-of-file before calling the **cread()** function.
- Use the **lseek()** function to determine the length of a file before calling **cread()**.
- Use the **_cread()** or **_creadv()** function to detect end-of-file or that the number of bytes read is less than the number of bytes requested.

Return Values

Upon successful completion, the **cread()** and **creadv()** functions return control to the calling process without returning a value. Otherwise **cread()** and **creadv()** print an error message to standard error and cause the calling process to terminate.

On success **_cread()** and **_creadv()** return the number of bytes read. On failure **cread()** and **_creadv()** return -1 and set *errno* to indicate the error. These functions return 0 (zero) if end-of-file is reached.

Errors

If the **_cread** or **_creadv** functions fail, *errno* may be set to one of the error code values set by the standard Unix **close()**, **lseek()**, **open()**, and **read()** functions or to the following values.

EPFSLAUTH PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.

EPFSLBADF PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.

EPFSLBADFH PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.

EPFSLBADID PFSLib: Bad I/O id. The I/O id sent to the server is incorrect.

EPSFLINVAL PFSLib: Invalid argument. Invalid argument sent to the server.

EPFSLIOLEN PFSLib: Too large I/O operation. The number of bytes in the operation is higher than PFSLib's maximum value.

EPFSLMIXIO PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.

EPFSLMREQUEST PFSLib: Too many outstanding request. There is no free I/O id left.

EPFSLNBYTES PFSLib: Read or written to few data. The call read less bytes than requested.

EPFSLNOIOD PFSLib: Couldn't get an I/O daemon. The server could not find an **iod** for the job.

EPFSREMOTE PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

cwrite(3 PFSLib), iread(3 PFSLib), iseof(3 PFSLib), iwrite(3 PFSLib), setiomode(3 PFSLib), lseek(3 PFSLib), open(3 PFSLib), read(2)

cwrite() (3 PFSLib) ____________ PFSLib V1_0 ____________ cwrite() (3 PFSLib)

Name

cwrite() — *Writes to a shared file and blocks the calling process until the write completes. (Synchronous write)*

Synopsis

```
#include &lt;pfslib.h&gt;

void cwrite ( int FileDescriptor,
              char * Buffer,
              unsigned int NBytes )

long _cwrite ( int FileDescriptor,
               char * Buffer,
               unsigned int NBytes )

#include &lt;uio.h&gt;
#include &lt;pfslib.h&gt;

void cwritev ( int FileDescriptor,
               struct iovec * iov,
               int iovCount )

long _cwritev ( int FileDescriptor,
                struct iovec * iov,
                int iovCount )
```

Paramters

FileDescriptor	File descriptor of a shared file.
Buffer	Pointer to the buffer containing the data to be written.
NBytes	Number of bytes to write to the file.
iov	Pointer to an array of `struct iovec` structures that identifies the buffers containing the data to be written. The iovec structure is defines in `sys/uio.h`.
iovCount	Number of `iovec` structures pointed to by the *iov* parameter.

Description

Other than return values and additional error, the **cwrite()** and **cwrite()** functions are identical to the **write()** and **writev()** functions, respectively.

These are a synchronous calls. The calling process waits (blocks) until the write completes. Use the **iwrite()** and **iwritev()** functions to write to a file without blocking the calling process.

Return Values

Upon successful completion, the **cwrite()** and **cwritev()** functions return control to the calling process without returning a value. Otherwise **cwrite()** and **cwritev()** print an error message to standard error and cause the calling process to terminate.

On success **_cwrite()** and **_cwritev()** return the number of bytes written. On failure **cwrite()** and **_cwritev()** return -1 and set *errno* to indicate the error.

Errors

If the **_cwrite** or **_cwritev** functions fail, *errno* may be set to one of the error code values set by the standard Unix **close()**, **lseek()**, **open()**, and **write()** functions or to the following values.

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLBADID	PFSLib: Bad I/O id. The I/O id sent to the server is incorrect.
EPSFLINVAL	PFSLib: Invalid argument. Invalid argument sent to the server.
EPFSLIOLEN	PFSLib: Too large I/O operation. The number of bytes in the operation is higher than PFSLib's maximum value.
EPFSLMIXIO	PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.
EPFSLMREQUEST	PFSLib: Too many outstanding request. There is no free I/O id left.
EPFSLNBYTES	PFSLib: Read or written to few data. The call wrote less bytes than requested.
EPFSLNOIOD	PFSLib: Couldn't get an I/O daemon. The sever could not find an **iod** for the job.
EPFSLREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

cread(3 PFSLib), iread(3 PFSLib), iseof(3 PFSLib), iwrite(3 PFSLib), setiomode(3 PFSLib), lseek(3 PFSLib), open(3 PFSLib), write(2)

gopen() (3 PFSLib) ____________ PFSLib V1_0 ____________ gopen() (3 PFSLib)

Name

gopen() — *Performs a global open of a file for reading or writing, sets the I/O mode of the file, and performs a global synchronization.*

Synopsis

```
#include <fcntl.h>
#include <sys/types.h>
#include <pfslib.h>

int gopen ( char * FileName,
            int OpenFlags,
            int IOMode,
            mode_t Mode )

int _gopen ( char * FileName,
             int OpenFlags,
             int IOMode,
             mode_t Mode )
```

Paramters

FileName Pointer to a pathname of the file to be opened or created.

OpenFlags Specifies the type of access, special open processing, the type of update, and the initial state of the open file. See **open()**.

IOMode I/O mode to be assigned to the file. See **setiomode()**.

Mode Specifies the permissions of the file to be created. If the file already exists, this parameter is ignored. See **open()**.

Description

The **gopen()** function performs a global open call which synchronizes the processes; all processes opens the same file without issuing multiple I/O requests.

If the parameter *FileName* of a shared file does not begin with '/' (slash), the path name of the current working directory as returned by **getcwd** is prepended to the file name which is used by the **pfsd**.

Warning!

Be aware that a shared file will be opened by the **pfsd** program on the machine it is located on. If you use an absolute path name make sure it is accessible on the **pfsd**'s machine. If you use a file name which does not begin with '/' (slash), make sure the pathname returned by **getcwd()**, is accessible on the **psfd**'s machine.

Return Values

On success **gopen()** and **_gopen()** return a file descriptor of the shared file. On failure **gopen()** will print an error message to standard error and cause the calling process to terminate; **_gopen()** returns -1 and sets *error* to indicate the error.

Errors

If the **_gopen()** function fails, *errno* may be set to one of the error code values set by the standard Unix **fstat()**, **getcwd()**, **lseek()**, **malloc()**, and **open()** functions or to the following values.

ENAMETOOLONG Length of the file name string exceeds its maximum.

EPSFLINVAL PFSLib: Invalid argument. Invalid argument sent to the server.

EPFSLNDELAY	PFSLib: **O_NDELAY** is not supported. PFSLib does not support non-blocking I/O using the **O_NDELAY** flag. Use asynchronous I/O operations (e.g. **iread**()).
EPFSLMFILE	PFSLib: Too many open files. The sever has no more space left in its file table.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

close(3 PFSLib), getcwd(3), open(2), open(3 PFSLib), setiomode(3 PFSLib)

Bugs

As file descriptors of shared files are inherited by a process created with the **fork()** system call, file access might lead to errors since parent and child process use the same socket connection to the PFSLib daemon process.

iodone() (3 PFSLib) ———— PFSLib V1_0 ———— iodone() (3 PFSLib)

Name

iodone() — *Determines whether an asynchronous read or write operation is complete.*

Synopsis

#include <pfslib.h>

`long` **iodone** (`long` *IOIdentifier*)

`long` **_iodone** (`long` *IOIdentifier*)

Paramters

IOIdentifier Non-negative I/O id returned by an asynchronous read or write library call (e.g. **iread**() or **iwrite**()).

Description

The **iodone()** function determines whether the asynchronous read or write operation associated with the the *IOIdentifier* parameter is complete.

Return Values

On success **iodone** and **_iodone()** return 1 if the read or write operation is complete. If the operation is not yet complete they return 0 (zero). On failure **iodone()** prints an error message to standard error causes the calling process to terminate; **_iodone()** returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLBADID	PFSLib: Bad I/O id.
EPFSLCHLDEXIT	PFSLib: Asynchronous I/O process terminated unsuccessfuly.
EPFSLCHLDSIG	PFSLib: Asynchronous I/O process terminated by signal.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

iowait(3 PFSLib), iread(3 PFSLib), iwrite(3 PFSLib)

iomode() (3 PFSLib) —— PFSLib V1_0 —— iomode() (3 PFSLib)

Name

iomode() — *gets the I/O mode of a file.*

Synopsis

#include <pfslib.h>

`long` **iomode** (`int` *FileDescriptor*)

`long` **_iomode** (`int` *FileDescriptor*)

Paramters

FileDescriptor File descriptor of a shared file.

Description

The **iomode()** functions determines the current I/O mode of the file identified by the *FileDescriptor* parameter.

Return Values

On success **iomode()** and **_iomode()** return the current I/O mode of the file identified by the *FileDescriptor* parameter. On failure **iomode()** prints an error message to standard error and causes the calling process to terminate; **_iomode()** returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

gopen(3 PFSLib), setiomode(3 PFSLib)

iowait() (3 PFSLib) ———— PFSLib V1_0 ———— iowait() (3 PFSLib)

Name

iowait() — *Waits (blocks) until an asynchronous read or write operation completes.*

Synopsis

#include <pfslib.h>

`void` **iowait** (`long` *IOIdentifier*)

`long` **_iowait** (`long` *IOIdentifier*)

Paramters

IOIdentifier Non-negative I/O id returned by an asynchronous read or write library call (e.g. **iread**() or **iwrite**()).

Description

The **iowait()** function waits until an asynchronous read or write operation associated with the the *IOIdentifier* parameter completes.

Return Values

Upon successful completion, the **iowait()** function returns control to the calling process without returning a value. Otherwise **iowait()** prints an error message to standard error and causes the calling process to terminate.

On success **_iowait()** returns 0. On failure it returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLBADID	PFSLib: Bad I/O id.
EPFSLCHLDEXIT	PFSLib: Asynchronous I/O process terminated unsuccessfuly.
EPFSLCHLDSIG	PFSLib: Asynchronous I/O process terminated by signal.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

iodone(3 PFSLib), iread(3 PFSLib), iwrite(3 PFSLib)

iread() (3 PFSLib) PFSLib V1_0 iread() (3 PFSLib)

Name

iread() — *Reads from a shared file and returns immediately. (Asynchronous read)*

Synopsis

```
#include <pfslib.h>

long iread ( int FileDescriptor,
             char * Buffer,
             unsigned int NBytes )

long _iread ( int FileDescriptor,
              char * Buffer,
              unsigned int NBytes )

#include <uio.h>
#include <pfslib.h>

long ireadv ( int FileDescriptor,
              struct iovec * iov,
              int iovCount )

long _ireadv ( int FileDescriptor,
               struct iovec * iov,
               int iovCount )
```

Paramters

FileDescriptor	File descriptor of a shared file.
Buffer	Pointer to the buffer in which to store the data after it is read form the file.
NBytes	Number of bytes to read from the file.
iov	Pointer to an array of `iovec` structures that identifies the buffers into which the data read is placed. The iovec structure is defines in `sys/uio.h`.
iovCount	Number of `iovec` structures pointed to by the *iov* parameter.

Description

Other than the return values, additional errors, and the asynchronous behavior the **iread()** and **ireadv()** functions are identical to the **read()** and **readv()** functions, respectively.

These calls are asynchronous calls. They return to the calling process immediately; the calling process continues its execution while the read is being done. Use **iowait()** or **iodone()** to determine whether the read operation completed.

Use the **iseof()** function to detect end-of-file.

Warning!

The number of available I/O ids is limited to 20 per process. If there are more than 20 outstanding operations, the call will return with an error. Use **iowait()** ot **iodone()** to free I/O ids of completed I/O operations.

Return Values

Upon successful completion, the **iread**(), **_iread**(), **ireadv**(), and **_ireadv()** functions return and non-negative I/O id for use in **iodone()** and **iowait**(). On failure, **iread**() and **ireadv**() print an error message to standard error and cause the calling process to terminate; **iread()** and **_ireadv()** return -1 and set *errno* to indicate the error.

Errors

If the **_iread** or **_ireadv** functions fail, *errno* may be set to one of the error code values set by the standard Unix **close()**, **fork()**, **lseek()**, **malloc()**, **open()**, **read()**, and **shmget()** functions or to the following values.

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLBADID	PFSLib: Bad I/O id. The I/O id sent to the server is incorrect.
EPSFLINVAL	PFSLib: Invalid argument. Invalid argument sent to the server.
EPFSLIOLEN	PFSLib: Too large I/O operation. The number of bytes in the operation is higher than PFSLib's maximum value.
EPFSLMIXIO	PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.
EPFSLMREQUEST	PFSLib: Too many outstanding request. There is no free I/O id left.
EPFSLNOIOD	PFSLib: Couldn't get an I/O daemon. The sever could not find an **iod** for the job.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

cread(3 PFSLib), cwrite(3 PFSLib), gopen(3 PFSLib), open(3 PFSLib), iodone(3 PFSLib), iowait(3 PFSLib), iseof(3 PFSLib), iwrite(3 PFSLib), setiomode(3 PFSLib)

iseof() (3 PFSLib) — PFSLib V1_0 — iseof() (3 PFSLib)

Name

iseof() — *Determines whether the file pointer is at end-of-file.*

Synopsis

#include <pfslib.h>

`long` **iseof** (`int` *FileDescriptor*)

`long` **_iseof** (`int` *FileDescriptor*)

Paramters

FileDescriptor File descriptor of a shared file.

Description

Use the **iseof()** function to determine whether the file pointer is at end-of-file. This function blocks until all asynchronous requests made by the process to the file are processed.

Return Values

Upon successful completion the **iseof()** and **_iseof()** function returns 0 (zero) if the file pointer is not at end-of-file or 1 if the file pointer is at end-of-file. On failure **iseof()** prints an error message to standard error and causes the calling process to terminate.; **_iseof()** returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.

EPFSLBADF PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.

EPFSLBADFH PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.

EPFSREMOTE PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

cread(3 PFSLib), cwrite(3 PFSLib), iread(3 PFSLib), iwrite(3 PFSLib), lseek(3 PFSLib)

iwrite() (3 PFSLib) ______ PFSLib V1_0 ______ iwrite() (3 PFSLib)

Name

iwrite() — *Writes to a shared file and returns immediately. (Asynchronous write)*

Synopsis

```
#include <pfslib.h>

void iwrite ( int FileDescriptor,
              char * Buffer,
              unsigned int NBytes )

long _iwrite ( int FileDescriptor,
               char * Buffer,
               unsigned int NBytes )

#include <uio.h>
#include <pfslib.h>

void iwritev ( int FileDescriptor,
               struct iovec * iov,
               int iovCount )

long _iwritev ( int FileDescriptor,
                struct iovec * iov,
                int iovCount )
```

Paramters

FileDescriptor	File descriptor of a shared file.
Buffer	Pointer to the buffer containing the data to be written.
NBytes	Number of bytes to write to the file.
iov	Pointer to an array of `iovec` structures that identifies the buffers containing the data to be written. The iovec structure is defines in `sys/uio.h`.
iovCount	Number of `iovec` structures pointed to by the *iov* parameter.

Description

Other than the return values, additional errors, and the asynchronous behavior the **iwrite()** and **iwritev()** functions are identical to the **write()** and **writev()** functions, respectively.

These calls are asynchronous calls. They return to the calling process immediately; the calling process continues its execution while the write is being done. Use **iowait()** or **iodone()** to determine whether the read operation completed.

Warning!

The number of available I/O ids is limited to 20 per process. If there are more than 20 outstanding operations, the call will return with an error. Use **iowait()** ot **iodone()** to free I/O ids of completed I/O operations.

Return Values

Upon successful completion, the **iwrite()**, **_iwrite()**, **iwritev()**, and **_iwritev()** functions return and non-negative I/O id for use in **iodone()** and **iowait()**. On failure, **iwrite()** and **iwrite()** print an error message to standard error and cause the calling process to terminate; **_iwrite()** and **_iwritev()** return -1 and set *errno* to indicate the error.

Errors

If the **_iwrite** or **_iwritev** functions fail, *errno* may be set to one of the error code values set by the standard Unix **close**(), **fork**(), **lseek**(), **malloc**(), **open**(), and **write()** functions or to the following values.

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLBADID	PFSLib: Bad I/O id. The I/O id sent to the server is incorrect.
EPSFLINVAL	PFSLib: Invalid argument. Invalid argument sent to the server.
EPFSLIOLEN	PFSLib: Too large I/O operation. The number of bytes in the operation is higher than PFSLib's maximum value.
EPFSLMIXIO	PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.
EPFSLMREQUEST	PFSLib: Too many outstanding request. There is no free I/O id left.
EPFSLNOIOD	PFSLib: Couldn't get an I/O daemon. The sever could not find an **iod** for the job.
EPFSLREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

cread(3 PFSLib), cwrite(3 PFSLib), gopen(3 PFSLib), open(3 PFSLib), iodone(3 PFSLib), iowait(3 PFSLib), iseof(3 PFSLib), iread(3 PFSLib), setiomode(3 PFSLib)

lseek() (3 PFSLib) ——— PFSLib V1_0 ——— lseek() (3 PFSLib)

Name

lseek() — *Set the file pointer to the requested position.*

Synopsis

```
#include &lt;sys/types.h&gt;
#include &lt;pfslib.h&gt;

off_t pfslib_lseek ( int Filedescriptor,
                     off_t Offset,
                     int Whence )

#define lseek pfslib_lseek
```

Paramters

FileDescriptor	File descriptor of a single accessed or shared file.
Offset	The value, in bytes, to be used together with the *Whence* parameter to set the file pointer position.
Whence	Specifies how *Offset* affects the file position. The values for the *Whence* parameter are as follows:
	SEEK_SET Sets the file position to *Offset*.
	SEEK_CUR Sets the file position to the current position plus *Offset*.
	SEEK_END Sets the file position to end-of-file plus *Offset*.

Description

The **pfslib_lseek()** function sets the file position of a shared file or a regular Unix file. Other than additional errors, it is identical to the standard **lseek()** system call with the following exceptions when accessing a shared file.

If the I/O mode of the shared file is **M_GLOBAL**, **M_RECORD**, or **M_SYNC** , **pfslib_lseek()** synchronizes the processes and the requested file position must be the same for all processes.

For compatibility reasons to Intel's PFS, **lseek** is defined as a C preprocessor macro to overlay the C-library call.

Return Values

Upon successful completion **pfslib_leek()** returns the new position of the file pointer as measured in bytes from the beginning of the file. On failure it returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLMIXIO	PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

lseek(2)

lsize() (3 PFSLib) PFSLib V1_0 lsize() (3 PFSLib)

Name

lsize() — *Increases the size of a file*

Synopsis

```
#include <sys/types.h>
#include <pfslib.h>

long lsize ( int Filedescriptor,
             off_t Offset,
             int Whence )

long _lsize ( int Filedescriptor,
              off_t Offset,
              int Whence )
```

Paramters

FileDescriptor File descriptor of a shared file.

Offset The value, in bytes, to be used together with the *Whence* parameter to increase the file size.

Whence Specifies how *Offset* affects the file size. The values for the *Whence* parameter are defined in pfslib.h as follows:

SIZE_SET Sets the file size to the greater of the current size or *Offset*.

SIZE_CUR Sets the file size to the greater of the current size or the current location plus *Offset*.

SIZE_END Sets the file size to the greater of the current size or the current size plus *Offset*.

Description

The **lsize()** function increases the size of the file according to the *Offset* and *Whence* parameters.

This functions is merely included for compatibility with Intel's PFS.

Return Values

On success, **lsize()** and **_lsize()** return the new size of the file. On failure, **lsize** prints an error message to standard error causes the calling process to terminate; **_lsize()** returns -1 and sets *errno* to indicate the error.

Errors

If the **_lsize** function fails, *errno* may be set to one of the error code values set by the standard Unix **ftruncate()** function or to the following values.

EPFSLAUTH PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.

EPFSLBADF PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.

EPFSLBADFH PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.

EPSFLINVAL PFSLib: Invalid argument. Invalid argument sent to the server.

EPFSREMOTE PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

lseek(3 PFSLib)

open() (3 PFSLib) ———— PFSLib V1_0 ———— open() (3 PFSLib)

Name

open() — *Opens or creates a local or shared file for reading or writing*

Synopsis

```
#include <fcntl.h>
#include <sys/types.h>
#include <pfslib.h>

int pfslib_open ( char * FileName,
                  int OpenFlags,
                  mode_t Mode )

int _pfslib_open ( char * FileName,
                   int OpenFlags,
                   mode_t Mode,
                   int IOMode,
                   int NumberOfClients,
                   int MyNumber,
                   int GlobalFlag )

#define open pfslib_open
```

Paramters

FileName	Pointer to a pathname of the file to be opened or created.
OpenFlags	Specifies the type of access, special open processing, the type of update, and the initial state of the open file. See **open()**.
Mode	Specifies the permissions of the file to be created. If the file already exists, this parameter is ignored. See **open()**.
IOMode	I/O mode to be assigned to the file. See **setiomode()**.
NumberOfClients	Number of processes accessing the shared files.
Mynumber	The calling process' number within the accessing processes.
GlobalFlag	Flag which specifies whether the call is synchronizing.

Description

The **pfslib_open()** is identical to the standard **open()** function except for addition features.

If the pattern '###' matches somewhere in the *FileName* parameter a local file for the requesting process will be opened. The pattern ###' will be substituted by the number (and leading zeros) of the process within the parallel application as specified in **pfslib_init()**. For example opening "myfile.###" will open "myfile.000" for process 0, "myfile.001" for process 1, and so on. Subsequent file operations must be standard Unix calls.

If the pattern '###' is not within the filename and 'pfs' is a substring in *FileName* a file for parallel access will be opened. Subsequent file operations will be handled by PFSLib. If 'pfs' is not a substring in *FileName* a standard Unix open will be called and all subsequent file operations must be standard Unix calls. Use **gopen()** to open a shared file which does not contain 'pfs' in *FileName*.

The **_pfslib_open()** function allows opening shared files with other *NumberOfClients* and *MyNumber* parameters than specified in **pfslib_init**. The *IOMode* parameter only has effect if *GlobalFlag* is not equal 0 (zero). If *GlobalFlag* is not equal 0 (zero) **_pfslib_open()** is a synchronizing call. **gopen** and **pfslib_open()** are both based on **_pfslib_open()**.

If the parameter *FileName* of a shared file does not begin with '/' (slash), the path name of the current working directory as returned by **getcwd** is prepended to the file name which is used by the **pfsd**.

Warning!

> Be aware that a shared file will be opened by the **pfsd** program on the machine it is located on. If you use an absolute path name make sure it is accessible on the **pfsd**'s machine. If you use a file name which does not begin with '/' (slash), make sure the pathname returned by **getcwd()**, is accessible on the **psfd**'s machine.

For compatibility reasons to Intel's PFS, **open** is defined as a C preprocessor macro to overlay the C-library call.

Return Values

On success **pfslib_open()** return a file descriptor of a shared or local file; **_pfslib_open()** returns a file descriptor to a shared file.

On failure the functions return -1 and set *errno* to indicate the error.

Errors

If the **pfslib_open** or **_pfslib_open** functions fail, *errno* may be set to one of the error code values set by the standard Unix **fstat()**, **getcwd()**, **lseek()**, **malloc()**, and **open()** functions or to the following values.

ENAMETOOLONG	Length of the file name string exceeds its maximum.
EPSFLINVAL	PFSLib: Invalid argument. Invalid argument sent to the server.
EPFSLNDELAY	PFSLib: **O_NDELAY** is not supported. PFSLib does not support non-blocking I/O using the **O_NDELAY** flag. Use asynchronous I/O operations (e.g. **iread**()).
EPFSLMFILE	PFSLib: Too many open files. The sever has no more space left in its file table.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

pfslib_init(3 PFSLib), open(2), getcwd(3), gopen(3 PFSLib), setiomode(3 PFSLib)

Bugs

As file descriptors of shared files are inherited by a process created with the **fork()** system call, file access might lead to errors since parent and child process use the same socket connection to the PFSLib daemon process.

pfslib_init() (3 PFSLib) ________ PFSLib V1_0 ________ pfslib_init() (3 PFSLib)

Name

pfslib_init() — *Initializing parallel file access*

Synopsis

```
#include <pfslib.h>

void pfslib_init ( char * PfsdHostName,
                   int NumberOfClients,
                   int MyNumber )

void _pfslib_init ( char * PfsdHostName,
                    int NumberOfClients,
                    int MyNumber,
                    int SeverThreshold,
                    int ClientThreshold )
```

Paramters

PfsdHostName	Name of the host the **pfsd** resides on.
NumberOfClients	Number of processes in the application accessing the shared files.
Mynumber	The calling process' number within the application.
ServerThreshold	Number of bytes up to which the I/O operation will be handled by the **pfsd**. I/O operations with higher amount of data will be handled by an **iod**.
ClientThreshold	Number of bytes up to which the I/O operation will always be synchronous on the client side. Asynchronous I/O operations with higher amount of data will be handled by a forked child process.

Description

With **pfslib_init()** a process of an application using PFSLibinitializes parallel file access and connects itself as a client to **pfsd**. Due to the independence of any parallel programming environment the process has to identify itself uniquely within the processes of the application by the *MyNumber* parameter. It must be in the range of 0 to *NumberOfClients*-1.

Warning!

Every process has to call this function before any other PFSLiboperation can be executed.

With **_pfslib_init** the user can specify other than the default server and client thresholds for asynchronous operations.

Return Values

On success **pfslib_init**, and **_pfslib_init** return control to the calling process, otherwise the calling process will be terminated.

Errors

See also

pfsd(1 PFSLib)

pfslib_perror() (3 PFSLib) ——— PFSLib V1_0 ——— pfslib_perror() (3 PFSLib)

Name

pfslib_perror() — *Print an error message explaining an subroutine error.*

Synopsis

#include <pfslib.h> #include <pfslib_errno.h>

`void` **pfslib_perror** (`char *` *String*)

Paramters

String A string to be printed in the error message.

Description

The **pfslib_perror()** subroutine writes a message on the standard error output that describes the last error encountered by a system call or library call. The error message includes the *String* parameter followed by a : (colon), a blank, the message, and a new-line character. The error number is taken from the global variable *errno*. If the error code is within the range of standard errors **pfslib_perror** behaves like **perror**. If the error code is within the range of PFSLib errors, the description of the error will be printed. If an error occured on the **pfsd** or **iod** side, this will be stated and a description of the remote error will be printed. Due to the heterogeneity of PFSLib, the error number of the remote error is not available since it may differ from system to system.

See also

perror(3)

setiomode() (3 PFSLib) ________ PFSLib V1_0 ________ setiomode() (3 PFSLib)

Name

setiomode() — *Sets the I/O mode of a shared file.*

Synopsis

#include <pfslib.h>

```
void setiomode ( int FileDescriptor,
                 int IOMode )

void _setiomode ( int FileDescriptor,
                  int IOMode )
```

Paramters

FileDescriptor File descriptor of a shared file.

IOMode I/O mode to be set. Values of *IOMode* are as follows:

M_UNIX Each process has its own file pointer, file operations are performed in first-come, first-serve basis, and access is unrestricted. This mode is set by default.

M_LOG All processes share a single file pointer, file operations are performed in first-come, first-serve basis.

M_SYNC All processes share a single file pointer, file operations are performed in order by node number. Records may have variable length. File operations are synchronizing.

M_RECORD Each process has its own file pointer, file operations are performed in first-come, first-serve basis. However, records are stored in the file in order by node number. Records must be of a fixed length.

M_GLOBAL All processes share a single file pointer and must perform the same operations in the same order. Data of a write operations will be written only once. In a read operation all processes will read the same data. File operations are synchronizing.

Description

The **setiomode()** function changes the I/O mode of a shared file. It must be performed by all processes with the same parameters and synchronizes the processes.

See the Intel's Paragon Manual for a detailed description.

Return Values

On success **setiomode()** returns control to the calling process. Otherwise it prints an error message to standard error and causes the calling process to terminate.

Upon successful completion **_setiomode()** behaves identically to **setiomode()**. On failure **_setiomode()** returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.

EPFSLBADF PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.

EPFSLBADFH PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.

EPSFLINVAL PFSLib: Invalid argument. Invalid argument sent to the server.

EPFSLMIXIO PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.

EPFSREMOTE PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

cread(3 PFSLib), cwrite(3 PFSLib), iread(3 PFSLib), iwrite(3 PFSLib), iomode(3 PFSLib)

iod (8 PFSLib) —— PFSLib V1_0 —— iod (8 PFSLib)

Name

iod — *PFSLib basic I/O server*

Synopsis

iod

Description

The **iod** programm will be started by the **pfsd** program. It is not intended to be executed by the user.

See also

pfsd(1 PFSLib)