PIPE

Motilal Nehru National Institute of Technology Allahabad, Prayagraj

- **A FILE MANAGEMENT Primer**

  - File-management system calls

    allow you to manipulate the full collection of regular, directory, and special files, including:

    - disk-based files
    - terminals
    - printers
    - interprocess communication facilities, such as pipes and sockets

  - open() is initially used to access or create a file.

    If system call succeeds,
    it returns a small integer called *a file descriptor* that is used in subsequent I/O operations on that file.

- A typical sequence of events:

```
int  fd;  /*  File descriptor */
 …
fd = open( fileName, … );   /* Open file, return file descriptor */
if ( fd==-1 )  {  /* Deal with error condition */ }
…
fcntl( fd, … ); /* Set some I/O flags if necessary */
…
read( fd, … );  /* Read from file */
…
write( fd, …); /* Write to file */
…
lseek( fd, …); /* Seek within file */
…
close(fd);  /* Close the file, freeing file descriptor */
```

- When a process no longer needs to access an open file, it should close it using the "close" system call.

- All of a process' open files are automatically closed when the process terminates.

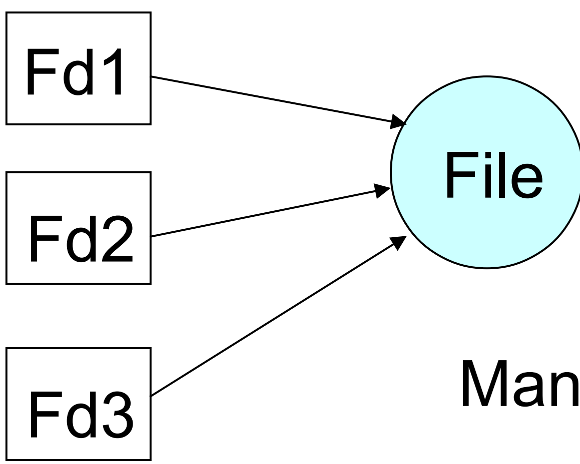
but, it's better programming practice to explicitly close your files.

- File descriptors are numbered sequentially, starting from zero. By convention, the first three file descriptor values have a special meaning:

Value	Meaning
0	standard input
1	standard output
2	standard error

- For Example,
  the "printf()" library function always sends its output using file descriptor 1.

  the "scanf()" always reads its input using file descriptor 0.



Many file descriptors, one file

- **File Pointer**

- A single file may be opened several times and, thus may have several file descriptors associated with it:

Each file descriptor has its own private set of properties:

A file pointer that records the offset in the file where it is reading and or writing.

When a file descriptor is created, its file pointer is positioned at offset 0 in the file (the first character) by default.

As the process reads and/or writes, the file pointer is updated accordingly.

- **THE MOST BASIC I/O SYSTEM CALLS**

Name	Function
open	opens/creates a file
read	reads bytes from a file into a buffer
write	writes bytes from a buffer to a file
lseek	moves to a particular offset in a file
close	closes a file
unlink	removes a file

- **Opening a File: open()**

System Call: int **open**(char* *fileName*, int *mode*[, int *permissions*])

"open()" allows you to open or create a file for reading and/or writing.

fileName : an absolute or relative pathname,

mode : a bitwise or'ing of a read/write flag together with zero or more miscellaneous flags.

permission : a number that encodes the value of the file's permission flags.

- The permissions value is affected by the process'**umask** value that we studied earlier.

- The read/write flags are as follows:

FLAG	MEANING
O_RDONLY	Open for read only.
O_WRONLY	Open for write only.
O_RDWR	Open for both read and write.

- The miscellaneous flags are as follows:

FLAG	MEANING
O_APPEND	Position the file pointer at the end of the file before each "write()".
O_CREAT	If the file doesn't exist, create the file and set the owner ID to the process' effective UID.
O_EXCL	If O_CREAT is set and the file exists, then "open()" fails.
O_TRUNC	If the file exists, it is truncated to length zero.

- **Creating a File**

- To create a file, use the O_CREAT flags as part of the mode flags and supply the initial file-permission flag settings as an octal value.

```
sprintf( tmpName, ".rev.%d", getpid() );  /* Random name */
/* Create temporary file to store copy of input */
tmpfd = open( tmpName, O_CREAT | O_RDWR, 0600);
if ( tmpfd== -1 ) fatalError();
```

- The "getpid()" function is a system call
  that returns the process'ID(PID), which is guaranteed to be unique.

- Files that begin with a period are sometimes known as *hidden files*.
  it doesn't show up in an **ls** listing.

- To open an existing file, specify the mode flags only.

```
fd = open( fileName, O_RDONLY );
if ( fd== -1 ) fatalError();
```

- **Reading From a File : read()**

To read bytes from a file, it uses the “read()” system call,

System Call: ssize_t read(int *fd*, void* *buf*, size_t *count*)

“read()” copies *count* bytes from the file referenced by the file descriptor fd into the buffer *buf*.

- if we read one character of input at a time,
  ⇨ a large number of system calls, thus slowing down the execution of our program considerably.

  ⇨ to read up to “BUFFER_SIZE” characters at a time.

```
charsRead = read( fd, buffer, BUFFER_SIZE );
if ( charsRead == 0 ) break;  /* EOF */
if ( charsRead == -1 ) fatalError();  /* Error */
```

- Writing to a File: write()

To write bytes to a file, it uses the "write()" system call, which works as follows:

System Call: ssize_t write(int fd, void* buf, size_t count)

"write()" copies *count* bytes from a buffer *buf* to the file referenced by the file descriptor *fd*.

If the O_APPEND flag was set for fd, the file position is set to the end of the file before each "write".

If successful, "write()" returns the number of bytes that were written; otherwise, it returns a value of -1.

- Perform the "write" operation:

```
/* Copy line to temporary file if reading standard input */

if ( standardInput )
   {
     charsWritten = write(tmpfd, buffer, charsRead );
     if ( charsWritten != charsRead ) fatalError();
   }
```

- Moving in a File : lseek()

System Call: off_t lseek(int *fd*, off_t *offset*, int *mode*)

"lseek()" allows you to change a descriptor's current file position.

fd : the file descriptor,
offset : a long integer,
mode : how offset should be interpreted.

- The three possible values of mode

VALUE	MEANING
SEEK_SET	offset is relative to the start of the file.
SEEK_CUR	offset is relative to the current file position.
SEEK_END	offset is relative to the end of the file.

- **The numbers of characters to read**

calculated by subtracting the offset value of the start of the next line from the offset value of the start of the current line:

```
lseek( fd, lineStart[i], SEEK_SET );  /* Find line and read it */
charsRead = read ( fd, buffer, lineStart[i+1] - lineStart[i] );
```

- To find out your current location without moving, use an offset value of zero relative to the current position:

```
currentOffset = lseek( fd, 0, SEEK_CUR );
```

```
$ cat sparse.c                    ---> list the test file.
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
/*****************************************/
main()
{
   int i, fd;
   /* Creates a sparse file */
   fd = open("sparse.txt", O_CREAT | O_RDWR, 0600 );
   write( fd, "space", 6);
   lseek( fd, 60006, SEEK_SET );
   write( fd, "file", 4 );
   close(fd);
   /* Create a normal file */
   fd = open("normal.txt", O_CREAT | O_RDWR, 0600 );
   write( fd, "normal", 6 );
   for ( i=1; i<=60000; i++ )
      write( fd, "/0", 1 );
```

```
    write( fd, "file", 4 );
    close(fd);
  }

$ sparse                 ---> execute the file.
$ ls -l  *.txt           ---> look at the files.
-rw-r--r--  1  glass       60010  Feb  14  15:06  normal.txt
-rw-r--r--  1  glass       60010  Feb  14  15:06  sparse.txt
$ ls  -s  *.txt          ---> list their block usage.
  60  normal.txt*        ---> uses a full 60 blocks.
   8  sparse.txt*        ---> only uses eight blocks.
$ _
```

- **Closing a File: “close()”**

- uses the “close()” system call to free the file descriptor of the input.

System Call : int close(int fd)

“close()” frees the file descriptor fd.

If fd is the last file descriptor associated with a particular open file, the kernel resources associated with the file are deallocated.

If successful, “close()” returns a value of 0;
otherwise, it returns a value of -1.

```
close(fd);   /* Close input file */
```

- **Deleting a File: unlink()**

System Call: int unlink(const char* *fileName*)

"unlink()" removes the hard link from the name *fileName* to its file.

If *fileName* is the last link to the file,
the file's resources are deallocated.

An executable file can "unlink" itself during execution
and still continue to completion.

If successful, "unlink()" returns a value of 0;
otherwise, it returns a value of -1.

```
If ( standardInput ) unlink( tmpName );  /* Remove temp file */
```

Jump To Process Management

- **Inodes**

  - UNIX uses a structure called an inode, which stands for “index node”, to store information about each file.

  - contains the locations of its disk blocks.

  - holds other information associated with a file, such as its permission flags, owner, group, and last modification time.

  - a structure of fixed size containing pointers to disk blocks and additional indirect pointers(for large files).

  - allocated a unique inode number, every file has exactly one inode.

  - stored in a special area at the start of the disk called the inode list.

- **Inode Contents**

- a list of the file information contained within each inode:
  - the type of the file : regular, directory, block special, character special, etc.
  - file permissions
  - the owner and group Ids
  - a hard-link count
  - the last modification and access times
  - if it’s a regular or directory file, the location of the blocks
  - if it’s a special file, the major and minor device numbers
  - if it’s a symbolic link, the value of the symbolic link

• Miscellaneous File Management System Calls

- a brief description of the following miscellaneous file management system calls:

NAME	Function
chown	changes a file’s owner and/or group
chmod	changes a file’s permission settings
dup	duplicates a file descriptor
dup2	similar to dup
fchown	works just like chown
fchmod	works just like chmod
fcntl	gives access to miscellaneous file characteristics
ftruncate	works just like truncate
ioctl	controls a device
link	creates a hard link
mknod	creates a special file
sync	schedulers all file buffers to be flushed to disk
truncate	truncates a file

- **Changing a File's Owner and/or Group: chown(), ichown() and fchown()**

- "chown()" and "fchown()" change the owner and/or group of a file and work like this:

```
System Call:  int chown( const char* fileName, uid_t ownerId,
                         gid_t  groupId )
              int lchown( const char* fileName, uid_t ownerId,
                          git_t  groupId )
              int fchown( int fd, uid_t ownerId, gid_t groupId )
```

- "chown()" causes the owner and group IDs of fileName to be changed to ownerId and groupId, respectively.

A value of -1 in a particular field means that its associated value should remain unchanged.

lchown():changes the ownership of a symbolic link
fchown():takes an open descriptor as an argument instead of a filename.

- **Changing a File's Owner and/or Group: chown(), ichown() and fchown()**

- Example, changed the group of the file "test.txt" from "music" to "cs". which has a group ID number of 62.

```
$ cat  mychown.c      ---> list the program.
main()
{
  int flag;
  flag = chown("test.txt", -1, 62 );  /* Leave user ID uchanged */
  if ( flag == -1 ) perror("mychown.c");
}

$ ls  -lg  test.txt       ---> examine file before the change.
-rw-r--r--   1   glass    music    3  May 25 11:42   test.txt
$ mychown                 ---> run program.
$ ls  -lg   test.txt      ---> examine file after the change.
-rw-r--r--   1   glass    cs       3  May 25  11:42   test.txt
$ -
```

- **Changing a File's Permissions: chmod() and fchmod()**

  - "chmod()" and "fchmod()" change a file's permission flags

```
System Call : int chmod( const char* fileName, int mode )
              int fchmod( int fd, mode_t mode );
```

"chmod()" changes the mode of *fileName* to *mode*,
where mode is usually supplied as an octal number,

To change a file's mode, you must either own it or be a super-user.

"fchmod()" works just like "chmod()" except that it takes an open file descriptor as an argument instead of a filename.

They both return a value of -1 if unsuccessful,
and a value of 0 otherwise.

• Changing a File's Permissions: chmod() and fchmod()

- changed the permission flags of the file "test.txt" to 600 octal, which corresponds to read and write permission for the owner only:

```
$ cat mychmod.c   ---> list the program.
main()
{
  int flag;
  flag = chmod("test.txt", 0600);  /* Use an octal encoding */
  if ( flag==-1 ) perror("mychmod.c");
}

$ ls -l test.txt    ---> examine file before the change.
-rw-r--r--    1   glass        3   May  25  11:42   test.txt
$ mychmod        ---> run the program.
$ ls -l test.txt    ---> examine file after the change.
-rw-------    1   glass        3   May  25  11:42   test.txt
$ -
```

• Duplicating a File Descriptor: dup() and dup2()

- "dup()", "dup2()"
  to duplicate file descriptors, and they work like this:

  System call: int dup(int *oldFd*)
  int dup2(int *oldFd*, int *newFd*)

  "dup()" finds the smallest free file-descriptor entry and points it to the same file to which *oldFd* points.

  "dup2()" closes *newFd* if it's currently active and then points it to the same file to which *oldFd* points.

- the original and copied file descriptors share the same file pointer and access mode.

- return the index of the new file descriptor if successful and a value of -1 otherwise.

- **Duplicating a File Descriptor: dup() and dup2()**

  - I created a file called "test.txt" and wrote to it via four different file descriptors:

    - The first file descriptor was the original descriptor.
    - The second descriptor was a copy of the first, allocated in slot 4.
    - The third descriptor was a copy of the first, allocated in slot 0 (the standard input channel), which was freed by the "close(0)" statement.
    - The fourth descriptor was a copy of the third descriptor, copied over the existing descriptor in slot 2 (the standard error channel).

- **Duplicating a File Descriptor: dup() and dup2()**

```
$ cat mydup.c     ---> list the file.
#include <stdio.h>
#include <fcntl.h>
main()
{
    int fd1, fd2, fd3;
    fd1 = open( “test.txt”, O_RDWR | O_TRUNC );
    printf(“fd1 = %d”\n”, fd1 );
    write( fd1, “what’s “, 6 );
    fd2 = dup(fd1);  /* Make a copy of fd1 */
    printf( “fd2=%d\n”, fd2);
    write( fd2, “up”, 3 );
    close(0);  /* Close standard input */
    fd3 = dup(fd1);  /* Make another copy of fd1 */
    printf(“fd3 = %d\n”, fd3);
    write(0, “ doc“, 4);
    dup2(3,2);  /* Duplicate channel 3 to channel 2 */
    write(2, “?\n”, 2 );
}
```

- **Duplicating a File Descriptor: dup() and dup2()**

```
$ mydup            ---> run the program.
fd1 = 3
fd2 = 4
fd3 = 0
$ cat test.txt     --->  list the output file.
what's up doc?
$ -
```

- **File Descriptor Operations: fcntl()**

  - "fcntl()" directly controls the settings of the flags associated with a file descriptor, and it works as follows:

System Call : int **fcntl**(int ***fd***, int ***cmd***, int ***arg***)

"fcntl()" performs the operation encoded by ***cmd*** on the file associated with the file descriptor ***fd***.

arg is an optional argument for ***cmd***.

- **File Descriptor Operations: fcntl()**

- Here are the most common values of cmd:

VALUE	OPERATION
F_SETFD	set the close-on-exec flag to the lowest bit of arg (0 or 1)
F_GETFD	return a number whose lowest bit is 1 if the close-on-exec flag is set and 0 otherwise.
F_GETFL	return a number corresponding to the current file status flags and access modes.
F_SETFL	set the current file-status flags to arg.
F_GETOWN	return the process ID or process group that is currently set to receive SIGIO/SIGURG signals.
F_SETOWN	set the process ID or process group that should receive SIGIO/SIGURG signals to arg.

- **File Descriptor Operations: fcntl()**

```
$ cat myfcntl.c          ---> list the program.
#include <stdio.h>
#include <fcntl.h>
main()
{
   int  fd;
   fd = open("test.txt", O_WRONLY );  /* Open file for writing */
   write( fd, "hi there\n", 9 );
   lseek( fd, 0, SEEK_SET );  /* Seek to beginning of file */
   fcntl(fd, F_SETFL, O_WRONLY | O_APPEND ); /* Set APPEND flag */
   write( fd, "   guys\n", 6 );
   close( fd );
}
$ cat test.txt          --->  list the original file.
here are the contents of
the original file.
$ myfcntl               --->  run the program.
```

- **File Descriptor Operations: fcntl()**

```
$ cat  test.txt          --->  list the new contents.
hi there
the contents of
the original file.
guys                    ---> note that “guys” is at the end.
$ -
```

- **Creating Special Files: mknod()**

- "mknod()" allows you to create a special file, and it works like this:

System Call: int mknod(const char* fileName, mode_t type,
dev_t device)

"mknod()" creates a new regular, directory, or special file called *fileName* whose type can be one of the following:

VALUE	MEANING
S_IFDIR	directory
S_IFCHR	character-oriented file
S_IFBLK	block-oriented file
S_IFREG	regular file
S_IFIFO	named pipe

- **Creating Special Files: mknod()**

- If the file is a character-or block-oriented file,

the low-order byte of device
should specify the minor device number,

the high-order byte
should specify the major device number.

In other cases, the value of device is ignored.

- Only a super-user can use “mknod()” to create directories,
character-oriented files, or block-oriented special files.

- It is typical now to use the “mkdir()” system call to create directories.
“mknod()” returns a value of -1 if unsuccessful
and a value of 0 otherwise.

- **Flushing the File-System Buffer: sync()**

  - "sync()" flushes the file-system buffers and works as follow:

System Call : void **sync**()

"sync()" schedules all of the file system buffers to be written to disk.

"sync()" should be performed by any programs that bypass the file system buffers and examine the raw file system.

"sync()" always succeeds.

- **Truncating a File: truncate() and ftruncate()**

- "truncate()" and "ftruncate()" set the length of a file,

System Call: int **truncate**(const char* *fileName*, off_t *length*)
int **ftruncate**(int fd, off_t *length*)

"truncate()" sets the length of the file *fileName* to be length bytes,
If the file is longer than *length*, it is truncated.

If it is shorter than *length*, it is padded with ASCII NULLS.

"ftruncate()" works just like "truncate()" does, except that it takes an open file descriptor as an argument instead of a filename.

They both return a value of -1 if unsuccessful
and a value of 0 otherwise.

- **Truncating a File: truncate() and ftruncate()**

```
$ cat  truncate.c     --->  list the program.
main()
{
   truncate("file1.txt", 10);
   truncate("file2.txt", 10);
}
$ cat  file1.txt       --->  list "file1.txt".
short
$ cat  file2.txt       --->  list "file2.txt".
long file with lots of letters
$ ls  -l   file*.txt   --->  examine both files.
-rw-r--r--   1  glass        6    May  25   12:16  file1.txt
-rw-r--r--   1  glass        32  May  25   12:17  file2.txt
$ truncate             ---> run the program.
$ ls  -l  file*.txt    ---> examine both files again.
-rw-r--r--   1  glass        10 May 25  12:16  file1.txt
-rw-r--r--   1  glass        10 May 25  12:17  file2.txt
```

- **Truncating a File: truncate() and ftruncate()**

```
$ cat  file1.txt        ---> "file1.txt"  is longer.
short
$ cat  file2.txt        ---> "file2.txt"  is shorter.
long file
$ -
```

- **IPC**

  - Interprocess Communication(IPC) is the generic term describing how two processes may exchange information with each other.

  - In general, the two processes may be running on the same machine or on different machines,

  although some IPC mechanisms may only support local usage (e.g., signals and pipes)

  - This communication may be an exchange of data for which two or more processes are cooperatively processing the data or synchronization information to help two independent,

  but related, processes schedule work so that they do not destructively overlap.

- **Pipes**

  - Pipes are an interprocess communication mechanism that allow two or more processes to send information to each other.

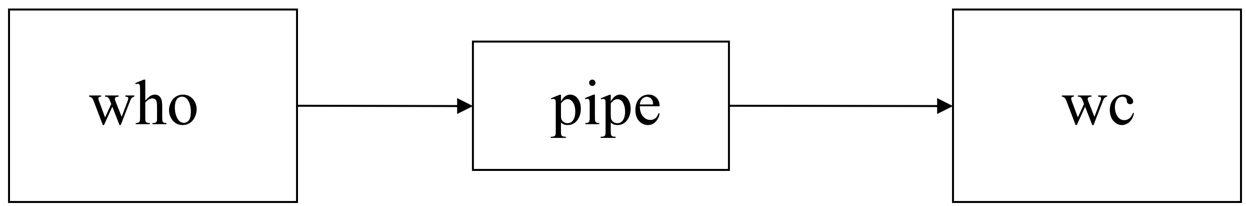
  - commonly used from within shells to connect the standard output of one utility to the standard input of another.

  - For example, here's a simple shell command that determines how many users there are on the system:

```
$ who | wc -l
```

  - The who utility generates one line of output per user. This output is then "piped" into the wc utility, which, when invoked with the "-l" option, outputs the total number of lines in its input.

- **Pipes**



Bytes from "who" flow through the pipe to "wc"

A simple pipe

- **Pipes**

  - It's important to realize that both the writer process and the reader process of a pipeline execute concurrently;

  - a pipe automatically buffers the output of the writer and suspends the writer if the pipe gets too full.

  - Similarly, if a pipe empties, the reader is suspended until some more output becomes available.

  - All versions of UNIX support unnamed pipes, which are the kind of pipes that shells use.

  - System V also supports a more powerful kind of pipe called a *named pipe*.

- **Unnamed Pipes: “pipe()”**

  - An unnamed pipe is a unidirectional communications link that automatically buffers its input (the maximum size of the input varies with different versions of UNIX, but is approximately 5K) and may be created using the “pipe()” system call.

  - Each end of a pipe has an associated file descriptor. The “write” end of the pipe may be written to using “write()”, and the “read” end may be read from using “read()”.

  - When a process has finished with a pipe’s file descriptor. it should close it using “close()”.

- **Unnamed Pipes: "pipe()"**

System Call : int **pipe**(int fd[2])

"pipe()" creates an unnamed pipe and returns two file descriptors:

The descriptor associated with the "read" end of the pipe is stored in fd[0],

and the descriptor associated with the "write" end of the pipe is stored in fd[1].

- If a process reads from a pipe whose "write" end has been closed, the "read()" call returns a value of zero, indicating the end of input.
- If a process reads from an empty pipe whose "write" end is still open, it sleeps until some input becomes available.

- **Unnamed Pipes: "pipe()"**

  - If a process tries to read more bytes from a pipe than are present, all of the current contents are returned and "read()" returns the number of bytes actually read.

  - If a process writes to a pipe whose "read" end has been closed, the write fails and the writer is sent a SIGPIPE signal. the default action of this signal is to terminate the receiver.

  - If a process writes fewer bytes to a pipe than the pipe can hold, the "write()" is guaranteed to be atomic; that is, the writer process will complete its system call without being preempted by another process.

  If the kernel cannot allocate enough space for a new pipe, "pipe()" returns a value of -1; otherwise, it returns a value of 0.

- **Unnamed Pipes: "pipe()"**

  - Assume that the following code was executed:

```
int fd[2];
pipe(fd);
```

the data structures shown in Figure 12.11 would be created.

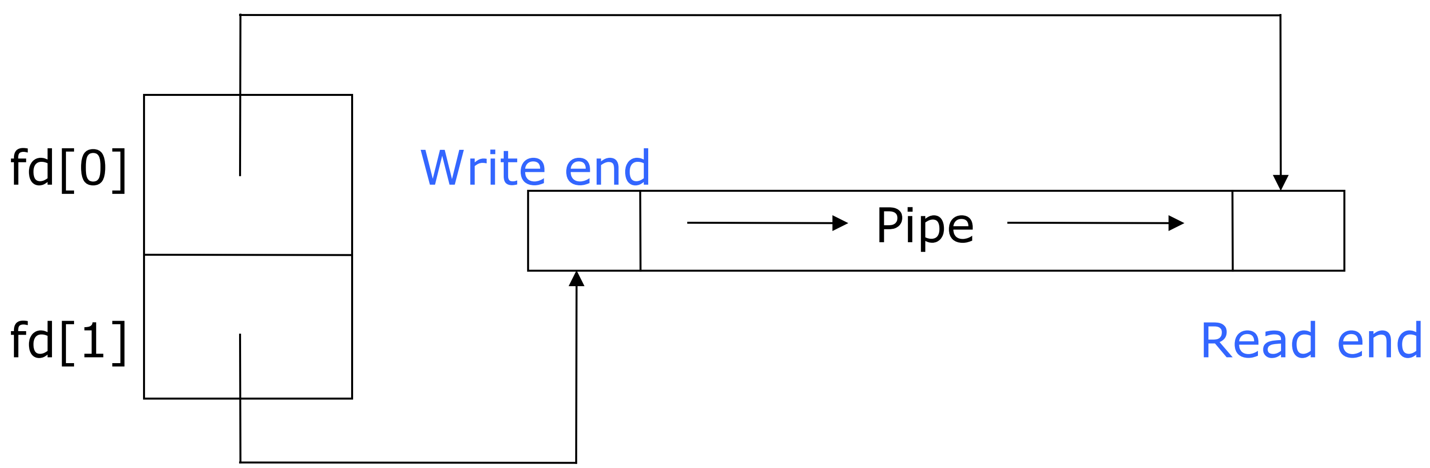


Figure 12.11 An unnamed pipe

- **Unnamed Pipes: "pipe()"**

  - Unnamed pipes are usually used for communication between a parent process and its child, with one process writing and the other process reading.

  The typical sequence of events for such a communication is as follows:

  1. The parent process creates an unnamed pipe using "pipe()".
  2. The parent process forks.
  3. The writer closes its "read" end of the pipe,
     and the designated reader closes its "write" end of the pipe.
  4. The processes communicate by using "write()" and "read()" calls.
  5. Each process closes its active pipe descriptor when it's finished
     with it.

- **Unnamed Pipes: “pipe()”**

  - Bidirectional communiation is only possible by using two pipes.

  Here’s a small program that uses a pipe to allow the parent to read a message from its child:

```
$ cat talk.c              ---> list the program.
#include <stdio.h>
#define  READ   0      /* The index of the “read” end of the pipe */
#define  WRITE  1      /* The index of the “write” end of the pipe */
char*  phrase =“Stuff this in your pipe and smoke it”;
main()
{
   int fd[2], bytesRead;
   char message[100];   /* Parent process’ message buffer */
   pipe(fd);  /* Create  an unnamed pipe */
```

- **Unnamed Pipes: "pipe()"**

```
if ( fork() == 0 )  /* Child, write */
  {
     close(fd[READ]); /* Close unused end */
     write(fd[WRITE], phrase, strlen(phrase)+1); /* Send */
     close(fd[WRITE]); /* Close used end */
  }
else      /* Parent, reader */
 {
    close(fd[WRITE]);  /* Close unused end */
    bytesRead = read( fd[READ], message, 100 ); /* Receive */
    printf("Read %d bytes: %s \n", bytesRead, message );
    close(fd[READ]);  /* Close used end */
  }
}
$ talk          ---> run the program.
Read 37 bytes: Stuff this in your pipe and smoke it
$ _
```

- **Unnamed Pipes: "pipe()"**

  - The child included the phrase's NULL terminator as part of the message so that the parent could easily display it.

  - When a writer process sends more than one variable-length message into a pipe, it must use a protocol to indicate to the reader the location for the end of the message.

  Methods for such indication include :

    - sending the length of a message(in bytes) before sending the message itself

    - ending a message with a special character such as a new line or a NULL

- **Unnamed Pipes: “pipe()”**

- UNIX shells use unnamed pipes to build pipelines.
connecting the standard output of the first to the standard input of the second.

```
$ cat  connect.c   ---> list the program.
#include <stdio.h>
#define  READ    0
#define  WRITE   1
main( argc, argv )
int argc;
char* argv[];
{
   int fd[2];
   pipe(fd);   /* Create an unnamed pipe */
   if ( fork()!=0 )  /* Parent, writer */
    {
      close( fd[READ] );  /* Close unused end */
```

- **Unnamed Pipes: “pipe()”**

```
        dup2( fd[WRITE], 1);  /* Duplicate used end to stdout */
        close( fd[WRITE] );    /* Close original used end */
        execlp( argv[1], argvp[1], NULL );  /* Execute writer program */
        perror( “connect” );    /* Should never execute */
    }
  else    /* Child, reader */
    {
        close( fd[WRITE] );    /* Close unused end */
        dup2( fd[READ], 0 );  /* Duplicate used end to stdin */
        close( fd[READ] );      /* Close original used end */
        execlp( argv[2], argv[2], NULL );  /* Execute reader program */
        perror( “connect” );    /* Should never execute */
    }
}
```

- **Unnamed Pipes: “pipe()”**

```
$ who              ---> execute “who” by itself.
gglass        ttyp0     Feb  15  18:15 (xyplex_3)

$ connect  who  wc      ---> pipe “who” through “wc”.
        1       6       57       …1 line,  6 words, 57 chars.
$ _
```

- **Named Pipes**

  - Named pipes, often referred to as FIFOs(first in, first out), are less restricted than unnamed pipes and offer the following advantages:

    - They have a name that exists in the file system.
    - They may be used by unrelated processes.
    - They exist until explicitly deleted.

  - Unfortunately, they are only supported by System V. named pipes have a larger buffer capacity, typically about 40K.

  - Named pipes exist as special files in the file system and may be created in one of two ways:

    - by using the UNIX mknod utility
    - by using the "mknod()" system call

- **Named Pipes**

  - To create a named pipe using mknod, use the “p” option. The mode of the named pipe may be set using chmod, allowing others to access the pipe that you create.

  Here’s an example of this procedure:

```
$ mknod myPipe  p            ---> create pipe.
$ chmod  ug+rw  myPipe   ---> update permissions.
$ ls -lg myPipe              ---> examine attributes.
prw-rw----   1   glass    cs         0  Feb  27  12:38  myPipe
$ _
```

- **Named Pipes**

  - To create a named pipe using "mknod()", specify "S_IFIFO" as the file mode.

    The mode of the pipe can then be changed using "chmod()".

  - C code that creates a name pipe with read and write permissions for the owner and group:

```
mknod("myPipe", SIFIFO, 0);    /* Create a named pipe */
chmod("myPipe", 0660);         /* Modify its permission flags */
```

  - Once a named pipe is opened using "open()", "write()" adds data at the start of the FIFO queue, and "read()" removes data from the end of the FIFO queue.

- **Named Pipes**

  - When a process has finished using a named pipe,
    it should close it using “close()”, and

  - when a named pipe is no longer needed,
    it should be removed from the file system using “unlink()”.

  - Like an unnamed pipe,
    a named pipe is intended only for use as a unidirectional link.

  - Writer processes should open a named pipe for writing only,
    and reader processes should open a pipe for reading only.

  Although a process can open a named pipe for both reading and writing, this usage doesn’t have much practical application.

- **Named Pipes**

  - an example program that uses named pipes,
  here are a couple of special rules concerning their use:

  - If a process tries to open a named pipe for reading only and no process currently has it open for writing,

    the reader will wait until a process opens it for writing, unless O_NONBLOCK or O_NDELAY is set, in which case "open()" succeeds immediately.

  - If a process tries to open a named pipe for writing only and no process currently has it open for reading,

    the writer will wait until a process opens it for reading, unless O_NONBLOCK or O_NDELAY is set, in which case "open()" fails immediately.

  - Named pipes will not work across a network.

- **Named Pipes**

  - The next examples uses two programs, "reader" and "writer", to demonstrate the use of named pipes,

    - A single reader process that creates a named pipe called "aPipe" is executed.

      It then reads and displays NULL-terminated lines from the pipe until the pipe is closed by all of the writing processes.

    - One or more writer processes are executed, each of which opens the named pipe called "aPipe" and sends three messages to it.

      If the pipe does not exist when a writer tries to open it, the writer retries every second until it succeeds.

      When all of a writer's messages are sent, the writer closes the pipe and exits.

- **Named Pipes**

  - Sample Output

```
$ reader & writer & writer &     ---> start 1 reader, 2 writers.
[1] 4698                         ---> reader process.
[2] 4699                         ---> first writer process.
[3] 4700                         ---> second writer process.
Hello from PID 4699
Hello from PID 4700
Hello from PID 4699
Hello from PID 4700
Hello from PID 4699
Hello from PID 4700
[2] Done   writer      ---> first writer exists.
[3] Done   writer      ---> second writer exists.
[4] Done   reader      ---> reader exists.
$ _
```

- **Named Pipes**

  - Reader Program

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>              /* For SIFIFO */
#include <fcntl.h>
/****************************************/
main()
{
   int fd;
   char str[100];
   unlink("aPipe");  /* Remove named pipe if it already exists */
   mknod("aPipe", S_IFIFO, 0);  /* Create name pipe */
   chmod("aPipe", 0660);  /* Change its permissions */
   fd = open("aPipe", O_RDONLY);  /* Open it for reading */
    while(readLine(fd, str) );  /* Display received messages */
      printf("%s\n", str);
   close(fd);   /* Close pipe */
 }
```

- **Named Pipes**

```
/****************************************/
readLine( fd, str )
int fd;
char* str;
/* Read s single NULL-terminated line into str from fd */
/* Return 0 when the end of input is reached and 1 otherwise */
{
  int n;
    do /* Read characters until NULL or end of input */
     {
        n = read( fd, str, 1);  /* Read one character */
     }
    while ( n>0 && *str++ != NULL );

  return ( n> 0 );  /* Return false if end of input */
 }
```

- **Named Pipes**

  - Writer Program

```
#include <stdio.h>
#include <fcntl.h>
/****************************************************/
main()
 {
  int fd, messageLen, i;
  char message[100];

  /* Prepare message */
  sprintf( message, "Hello from PID %d", getpid() );
  messageLen = strlen( message ) +1;
  do /* Keep trying to open the file until successful */
  {
    fd = open( "aPipe", O_WRONLY ); /*Open named pipe for writing */
    if ( fd == -1 ) sleep(1);  /* Try again in 1 second */
  } while ( fd == -1 );
```

- **Named Pipes**

```
for ( i=1; i<=3; i++)  /* Send three messages */
 {
   write( fd, message, messageLen );  /* Write message down pipe */
   sleep(3);   /* Pause a while */
 }
close(fd);   /* Close pipe descriptor */

}
```