UNIT 1
UNIX FILES

Files are the building blocks of any operating system. When you execute a command in UNIX, the UNIX kernel fetches the corresponding executable file from a file system, loads its instruction text to memory, and creates a process to execute the command on your behalf. In the course of execution, a process may read from or write to files. All these operations involve files. Thus, the design of an operating system always begins with an efficient file management system.

File Types

A file in a UNIX or POSIX system may be one of the following types:

- regular file
- directory file
- FIFO file
- Character device file
- Block device file

Regular file

- A regular file may be either a text file or a binary file
- These files may be read or written to by users with the appropriate access permission
- Regular files may be created, browsed through and modified by various means such as text editors or compilers, and they can be removed by specific system commands

Directory file

- It is like a folder that contains other files, including sub-directory files.
- It provides a means for users to organise their files into some hierarchical structure based on file relationship or uses.
- Ex: **/bin** directory contains all system executable programs, such as **cat, rm, sort**
- A directory may be created in UNIX by the **mkdir** command
  - Ex: `mkdir /usr/foo/xyz`
- A directory may be removed via the **rmdir** command
  - Ex: `rmdir /usr/foo/xyz`
- The content of directory may be displayed by the **ls** command

Device file

Block device file	Character device file
It represents a physical device that transmits data a block at a time.	It represents a physical device that transmits data in a character-based manner.
Ex: hard disk drives and floppy disk drives	Ex: line printers, modems, and consoles

- A physical device may have both block and character device files representing it for different access methods.
- An application program may perform read and write operations on a device file and the OS will automatically invoke an appropriate device driver function to perform the actual data transfer between the physical device and the application
- An application program in turn may choose to transfer data by either a character-based(via character device file) or block-based(via block device file)
- A device file is created in UNIX via the **mknod** command
  - Ex: `mknod        /dev/cdsk      c     115     5`

Here ,

c	-	character device file
115	-	major device number
5	-	minor device number

    - For block device file, use argument ‘b’ instead of ‘c’.
- **Major device number** → an index to a kernel table that contains the addresses of all device driver functions known to the system. Whenever a process reads data from or writes data to a device file, the kernel uses the device file’s major number to select and invoke a device driver function to carry out actual data transfer with a physical device.
- **Minor device number** → an integer value to be passed as an argument to a device driver function when it is called. It tells the device driver function what actual physical device is talking to and the I/O buffering scheme to be used for data transfer.

❖ **FIFO file**

- It is a special pipe device file which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer.
- The size of the buffer is fixed to PIPE_BUF.
- Data in the buffer is accessed in a first-in-first-out manner.
- The buffer is allocated when the first process opens the FIFO file for read or write
- The buffer is discarded when all processes close their references (stream pointers) to the FIFO file.
- Data stored in a FIFO buffer is temporary.
- A FIFO file may be created via the **mkfifo** command.
    - The following command creates a FIFO file (if it does not exists)

```
mkfifo   /usr/prog/fifo_pipe
```

    - The following command creates a FIFO file (if it does not exists)

```
mknod    /usr/prog/fifo_pipe         p
```

- FIFO files can be removed using **rm** command.

❖ **Symbolic link file**

- BSD UNIX & SV4 defines a symbolic link file.
- A symbolic link file contains a path name which references another file in either local or a remote file system.
- POSIX.1 does not support symbolic link file type
- A symbolic link may be created in UNIX via the **ln** command
- Ex: `ln    -s    /usr/divya/original         /usr/raj/slink`
- It is possible to create a symbolic link to reference another symbolic link.
- **rm, mv** and **chmod** commands will operate only on the symbolic link arguments directly and not on the files that they reference.

The UNIX and POSIX File Systems

- Files in UNIX or POSIX systems are stored in tree-like hierarchical file system.
- The root of a file system is the root (“/”) directory.
- The leaf nodes of a file system tree are either empty directory files or other types of files.
- **Absolute path name** of a file consists of the names of all the directories, starting from the root.
- **Ex:** **/usr/divya/a.out**
- **Relative path name** may consist of the “**.**” and “**..**” characters. These are references to current and parent directories respectively.
- **Ex:** **../../.login** denotes .login file which may be found 2 levels up from the current directory
- A file name may not exceed NAME_MAX characters (14 bytes) and the total number of characters of a path name may not exceed PATH_MAX (1024 bytes).
- POSIX.1 defines _POSIX_NAME_MAX and _POSIX_PATH_MAX in <limits.h> header
- File name can be any of the following character set only

A to Z	a to z	0 to 9	_

- Path name of a file is called the **hardlink.**
- A file may be referenced by more than one path name if a user creates one or more hard links to the file using **ln** command.

```
ln    /usr/foo/path1          /usr/prog/new/n1
```

- If the –s option is used, then it is a symbolic (soft) link .

The following files are commonly defined in most UNIX systems

FILE	Use
/etc	Stores system administrative files and programs
/etc/passwd	Stores all user information's
/etc/shadow	Stores user passwords
/etc/group	Stores all group information
/bin	Stores all the system programs like cat, rm, cp,etc.
/dev	Stores all character device and block device files
/usr/include	Stores all standard header files.
/usr/lib	Stores standard libraries
/tmp	Stores temporary files created by program

The UNIX and POSIX File Attributes

The general file attributes for each file in a file system are:

1) File type - specifies what type of file it is.
2) Access permission - the file access permission for owner, group and others.
3) Hard link count - number of hard link of the file
4) Uid - the file owner user id.
5) Gid - the file group id.
6) File size - the file size in bytes.
7) Inode no - the system inode no of the file.
8) File system id - the file system id where the file is stored.
9) Last access time - the time, the file was last accessed.
10) Last modified time - the file, the file was last modified.
11) Last change time - the time, the file was last changed.

In addition to the above attributes, UNIX systems also store the major and minor device numbers for each device file. All the above attributes are assigned by the kernel to a file when it is created. The attributes that are constant for any file are:

- ✓ File type
- ✓ File inode number
- ✓ File system ID
- ✓ Major and minor device number

The other attributes are changed by the following UNIX commands or system calls

Unix Command	System Call	Attributes changed
chmod	**chmod**	Changes access permission, last change time
chown	**chown**	Changes UID, last change time
chgrp	**chown**	Changes GID, ast change time
touch	**utime**	Changes last access time, modification time
ln	**link**	Increases hard link count
rm	**unlink**	Decreases hard link count. If the hard link count is zero, the file will be removed from the file system
vi, emac		Changes the file size, last access time, last modification time

Inodes in UNIX System V

- In UNIX system V, a file system has an inode table, which keeps tracks of all files. Each entry of the inode table is an inode record which contains all the attributes of a file, including inode # and the physical disk address where data of the file is stored
- For any operation, if a kernel needs to access information of a file with an inode # 15, it will scan the inode table to find an entry, which contains an inode # 15 in order to access the necessary data.
- An inode # is unique within a file system. A file inode record is identified by a file system ID and an inode #.
- Generally an OS does not keep the name of a file in its record, because the mapping of the filenames to inode# is done via directory files i.e. a directory file contains a list of names of their respective inode # for all file stored in that directory.
- Ex: a sample directory file content

Inode number	File name
115	
89	..
201	xyz
346	a.out
201	xyz_ln1

- To access a file, for example /usr/divya, the UNIX kernel always knows the "/" (root) directory inode # of any process. It will scan the "/" directory file to find the inode number of the usr file. Once it gets the usr file inode #, it accesses the contents of usr file. It then looks for the inode # of divya file.
- Whenever a new file is created in a directory, the UNIX kernel allocates a new entry in the inode table to store the information of the new file
- It will assign a unique inode # to the file and add the new file name and inode # to the directory file that contains it.

Application Program Interface to Files

The general interfaces to the files on UNIX and POSIX system are

- Files are identified by pathnames.
- Files should be created before they can be used. The various commands and system calls to create files are listed below.

File type	commands	system call
Regular file	vi,pico,emac	open,creat
Directory file	mkdir	mkdir,mknod
FIFO file	mkfifo	mkfifo,mknod
Device file	mknod	mknod
Symbolic link file	ln –s	symlink

- For any application to access files, first it should be opened, generally we use **open** system call to open a file, and the returned value is an integer which is termed as file descriptor.
- There are certain limits of a process to open files. A maximum number of OPEN-MAX files can be opened .The value is defined in <limits.h> header
- The data transfer function on any opened file is carried out by **read** and **write** system call.
- File hard links can be increased by **link** system call, and decreased by **unlink** system call.
- File attributes can be changed by **chown**, **chmod** and **link** system calls.
- File attributes can be queried (found out or retrieved) by **stat** and **fstat** system call.
- UNIX and POSIX.1 defines a structure of data type stat i.e. defined in <sys/stat.h> header file. This contains the user accessible attribute of a file. The definition of the structure can differ among implementation, but it could look like

```
struct stat
{
        dev_t  st_dev;         /* file system ID */
        ino_t  st_ino;         /* file inode number */
        mode_t        st_ mode;   /* contains file type and permission */
        nlink_t       st_nlink;   /* hard link count */
        uid_t  st_uid;         /* file user ID */
        gid_t  st_gid;         /* file group ID */
        dev_t  st_rdev;        /*contains major and minor device#*/
        off_t  st_size;    /* file size in bytes */
        time_t        st_atime;  /* last access time */
        time_t        st_mtime;     /* last modification time */
        time_t        st_ctime;     /* last status change time */
};
```

UNIX Kernel Support for Files

In UNIX system V, the kernel maintains a file table that has an entry of all opened files and also there is an inode table that contains a copy of file inodes that are most recently accessed.

A process, which gets created when a command is executed will be having its own data space (data structure) wherein it will be having file descriptor table. The file descriptor table will be having an maximum of OPEN_MAX file entries. Whenever the process calls the **open** function to open a file to read or write, the kernel will resolve the pathname to the file inode number.

The steps involved are :

1. The kernel will search the process descriptor table and look for the first unused entry. If an entry is found, that entry will be designated to reference the file .The index of the entry will be returned to the process as the file descriptor of the opened file.

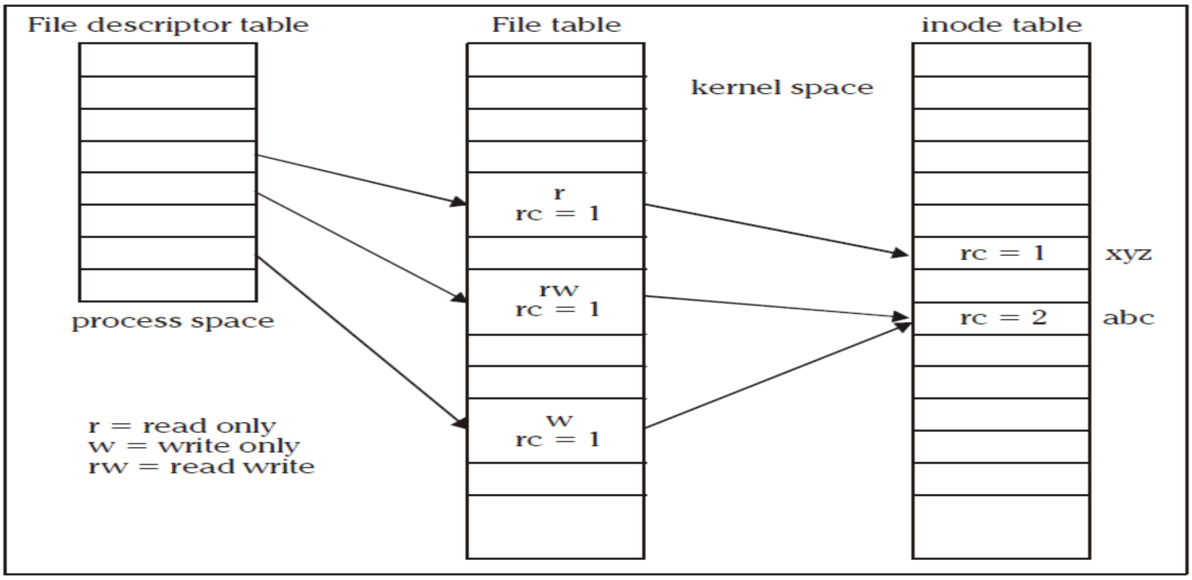
2. The kernel will scan the file table in its kernel space to find an unused entry that can be assigned to reference the file.

If an unused entry is found the following events will occur:

- The process file descriptor table entry will be set to point to this file table entry.
- The file table entry will be set to point to the inode table entry, where the inode record of the file is stored.
- The file table entry will contain the current file pointer of the open file. This is an offset from the beginning of the file where the next read or write will occur.
- The file table entry will contain an open mode that specifies that the file opened is for read only, write only or read and write etc. This should be specified in open function call.
- The reference count (rc) in the file table entry is set to 1. Reference count is used to keep track of how many file descriptors from any process are referring the entry.
- The reference count of the in-memory inode of the file is increased by 1. This count specifies how many file table entries are pointing to that inode.

If either (1) or (2) fails, the **open** system call returns -1 (failure/error)

Data Structure for File Manipulation



Normally the reference count in the file table entry is 1,if we wish to increase the rc in the file table entry, this can be done using fork,dup,dup2 system call. When a open system call is succeeded, its return value will be an integer (file

descriptor). Whenever the process wants to read or write data from the file, it should use the file descriptor as one of its argument.

The following events will occur whenever a process calls the **close** function to close the files that are opened.

1. The kernel sets the corresponding file descriptor table entry to be unused.

2. It decrements the rc in the corresponding file table entry by 1, if rc not equal to 0 go to step 6.

3. The file table entry is marked as unused.

4. The rc in the corresponding file inode table entry is decremented by 1, if rc value not equal to 0 go to step 6.

5. If the hard link count of the inode is not zero, it returns to the caller with a success status otherwise it marks the inode table entry as unused and de-allocates all the physical dusk storage of the file.

6. It returns to the process with a 0 (success) status.

Directory Files

- It is a record-oriented file
- Each record contains the information of a file residing in that directory
- The record data type is *struct dirent* in UNIX System V and POSIX.1 and *struct direct* in BSD UNIX.
- The record content is implementation-dependent
- They all contain 2 essential member fields
    - File name
    - Inode number
- Usage is to map file names to corresponding inode number

Directory function	Purpose
opendir	Opens a directory file
readdir	Reads next record from the file
closedir	Closes a directory file
rewinddir	Sets file pointer to beginning of file

Hard and Symbolic Links

- A hard link is a UNIX pathname for a file. Generally most of the UNIX files will be having only one hard link.
- In order to create a hard link, we use the command **ln**.

  Example : Consider a file `/usr/ divya/old`, to this we can create a hard link by

  ```
  ln     /usr/ divya/old              /usr/ divya/new
  ```

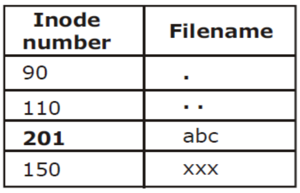
  after this we can refer the file by either `/usr/ divya/old` or `/usr/ divya/new`
- Symbolic link can be creates by the same command ln but with option –s

  Example: `ln   –s   /usr/divya/old    /usr/divya/new`
- **ln** command differs from the **cp**(copy) command in that **cp** creates a duplicated copy of a file to another file with a different pathname, whereas **ln** command creates a new directory to reference a file.
- Let's visualize the content of a directory file after the execution of command **ln**.

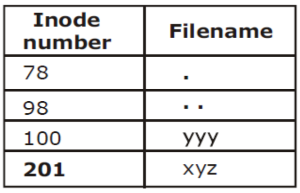
  Case 1: for hardlink file

  ```
  ln    /usr/divya/abc          /usr/raj/xyz
  ```

  The content of the directory files `/usr/divya` and `/usr/raj` are



Inode number	Filename
90	.
110	..
201	abc
150	xxx



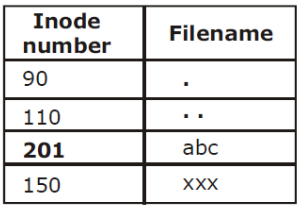
Inode number	Filename
78	.
98	..
100	yyy
201	xyz

Both `/urs/divya/abc` and `/usr/raj/xyz` refer to the same inode number 201, thus type is no new file created.

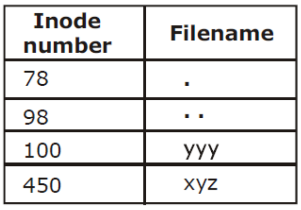
Case 2: For the same operation, if ln –s command is used then a new inode will be created.

```
ln –s     /usr/divya/abc   /usr/raj/xyz
```

The content of the directory files divya and raj will be



Inode number	Filename
90	.
110	..
201	abc
150	xxx



Inode number	Filename
78	.
98	..
100	yyy
450	xyz

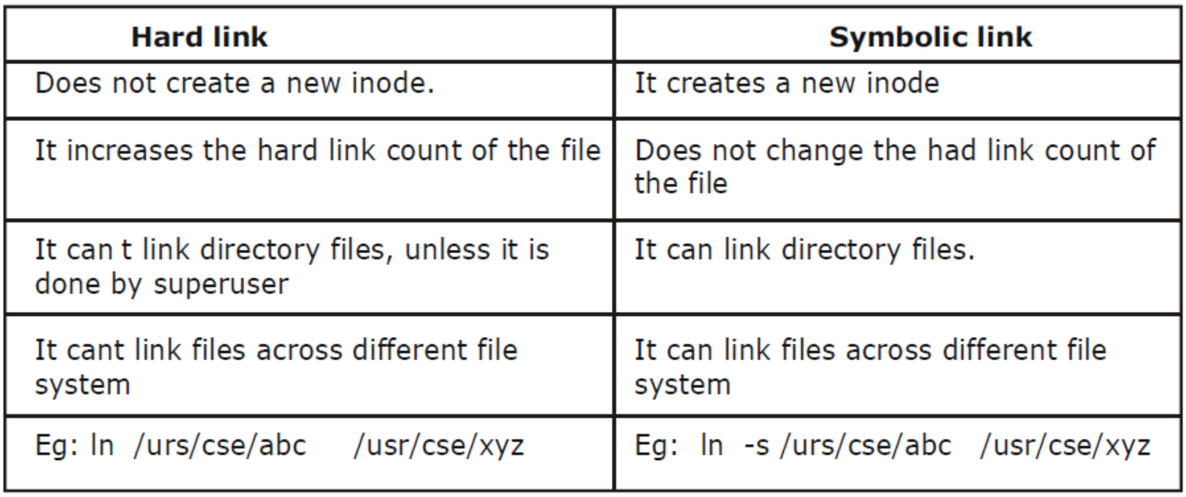
If cp command was used then the data contents will be identical and the 2 files will be separate objects in the file system, whereas in ln –s the data will contain only the path name.

Limitations of hard link:

1. User cannot create hard links for directories, unless he has super-user privileges.

2. User cannot create hard link on a file system that references files on a different file system, because inode number is unique to a file system.

Differences between hard link and symbolic link are listed below:



Hard link	Symbolic link
Does not create a new inode.	It creates a new inode
It increases the hard link count of the file	Does not change the had link count of the file
It can t link directory files, unless it is done by superuser	It can link directory files.
It cant link files across different file system	It can link files across different file system
Eg: ln /urs/cse/abc /usr/cse/xyz	Eg: ln -s /urs/cse/abc /usr/cse/xyz

UNIX FILE APIs

General file API's

Files in a UNIX and POSIX system may be any one of the following types:

- Regular file
- Directory File
- FIFO file
- Block device file
- character device file
- Symbolic link file.

There are special API's to create these types of files. There is a set of Generic API's that can be used to manipulate and create more than one type of files. These API's are:

open

- ✓ This is used to establish a connection between a process and a file i.e. it is used to open an existing file for data transfer function or else it may be also be used to create a new file.
- ✓ The returned value of the open system call is the file descriptor (row number of the file table), which contains the inode information.
- ✓ The prototype of open function is

```
#include<sys/types.h>
#include<sys/fcntl.h>
int open(const char *pathname, int accessmode, mode_t permission);
```

- ✓ If successful, open returns a nonnegative integer representing the open file descriptor.
- ✓ If unsuccessful, open returns –1.
- ✓ The first argument is the name of the file to be created or opened. This may be an absolute pathname or relative pathname.
- ✓ If the given pathname is symbolic link, the open function will resolve the symbolic link reference to a non symbolic link file to which it refers.
- ✓ The second argument is access modes, which is an integer value that specifies how actually the file should be accessed by the calling process.
- ✓ Generally the access modes are specified in <fcntl.h>. Various access modes are:

O_RDONLY	- open for reading file only
O_WRONLY	- open for writing file only
O_RDWR	- opens for reading and writing file.

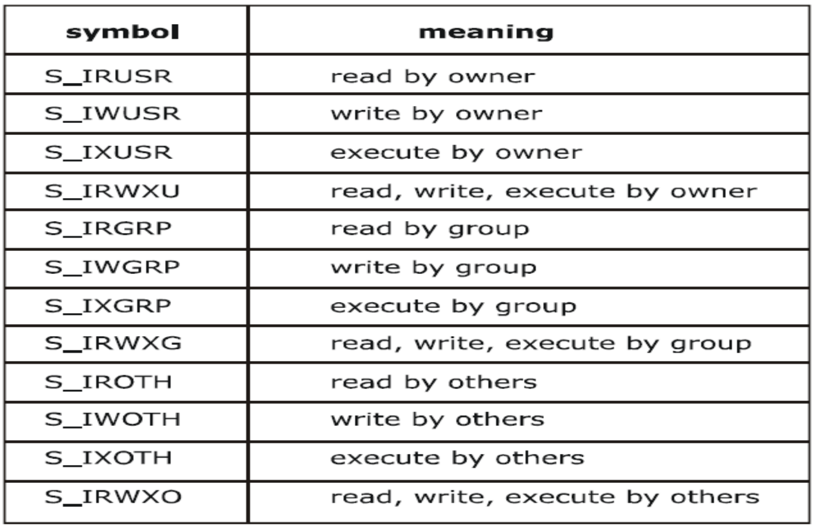
There are other access modes, which are termed as access modifier flags, and one or more of the following can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file.

O_APPEND	- Append data to the end of file.
O_CREAT	- Create the file if it doesn't exist
O_EXCL	- Generate an error if O_CREAT is also specified and the file already exists.
O_TRUNC	- If file exists discard the file content and set the file size to zero bytes.
O_NONBLOCK	- Specify subsequent read or write on the file should be non-blocking.
O_NOCTTY	- Specify not to use terminal device file as the calling process control terminal.

- ✓ To illustrate the use of the above flags, the following example statement opens a file called /usr/divya/usp for read and write in append mode:

```
int fd=open(“/usr/divya/usp”,O_RDWR | O_APPEND,0);
```

- ✓ If the file is opened in read only, then no other modifier flags can be used.
- ✓ If a file is opened in write only or read write, then we are allowed to use any modifier flags along with them.
- ✓ The third argument is used only when a new file is being created. The symbolic names for file permission are given in the table in the previous page.



symbol	meaning
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute by owner
S_IRWXU	read, write, execute by owner
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXG	read, write, execute by group
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute by others
S_IRWXO	read, write, execute by others

creat

- This system call is used to create new regular files.
- The prototype of creat is

```
#include <sys/types.h>
#include<unistd.h>
int creat(const char *pathname, mode_t mode);
```

- Returns: file descriptor opened for write-only if OK, -1 on error.
- The first argument pathname specifies name of the file to be created.
- The second argument mode_t, specifies permission of a file to be accessed by owner group and others.
- The creat function can be implemented using open function as:

```
#define creat(path_name, mode)
open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

read

- The read function fetches a fixed size of block of data from a file referenced by a given file descriptor.
- The prototype of read function is:

```
#include<sys/types.h>
#include<unistd.h>
size_t read(int fdesc, void *buf, size_t nbyte);
```

- If successful, read returns the number of bytes actually read.
- If unsuccessful, read returns –1.
- The first argument is an integer, fdesc that refers to an opened file.
- The second argument, buf is the address of a buffer holding any data read.
- The third argument specifies how many bytes of data are to be read from the file.
- The size_t data type is defined in the <sys/types.h> header and should be the same as unsigned int.
- There are several cases in which the number of bytes actually read is less than the amount requested:
    - When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
    - When reading from a terminal device. Normally, up to one line is read at a time.
    - When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
    - When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.

write

- The write system call is used to write data into a file.
- The write function puts data to a file in the form of fixed block size referred by a given file descriptor.

- ✓ The prototype of write is

```
#include<sys/types.h>
#include<unistd.h>
ssize_t write(int fdesc, const void *buf, size_t size);
```

- ✓ If successful, write returns the number of bytes actually written.
- ✓ If unsuccessful, write returns –1.
- ✓ The first argument, fdesc is an integer that refers to an opened file.
- ✓ The second argument, buf is the address of a buffer that contains data to be written.
- ✓ The third argument, size specifies how many bytes of data are in the buf argument.
- ✓ The return value is usually equal to the number of bytes of data successfully written to a file. (*size* value)

❖ close

- ✓ The close system call is used to terminate the connection to a file from a process.
- ✓ The prototype of the close is

```
#include<unistd.h>
int close(int fdesc);
```

- ✓ If successful, close returns 0.
- ✓ If unsuccessful, close returns –1.
- ✓ The argument fdesc refers to an opened file.
- ✓ Close function frees the unused file descriptors so that they can be reused to reference other files. This is important because a process may open up to OPEN_MAX files at any time and the close function allows a process to reuse file descriptors to access more than OPEN_MAX files in the course of its execution.
- ✓ The close function de-allocates system resources like file table entry and memory buffer allocated to hold the read/write.

❖ fcntl

- ✓ The fcntl function helps a user to query or set flags and the close-on-exec flag of any file descriptor.
- ✓ The prototype of fcntl is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd, …);
```

- ✓ The first argument is the file descriptor.
- ✓ The second argument cmd specifies what operation has to be performed.
- ✓ The third argument is dependent on the actual cmd value.
- ✓ The possible cmd values are defined in <fcntl.h> header.

cmd value	Use
F_GETFL	Returns the access control flags of a file descriptor fdesc
F_SETFL	Sets or clears access control flags that are specified in the third argument to fcntl. The allowed access control flags are O_APPEND & O_NONBLOCK
F_GETFD	Returns the close-on-exec flag of a file referenced by fdesc. If a return value is zero, the flag is off; otherwise on.
F_SETFD	Sets or clears the close-on-exec flag of a fdesc. The third argument to fcntl is an integer value, which is 0 to clear the flag, or 1 to set the flag
F_DUPFD	Duplicates file descriptor fdesc with another file descriptor. The third argument to fcntl is an integer value which specifies that the duplicated file descriptor must be greater than or equal to that value. The return value of fcntl is the duplicated file descriptor

- ✓ The fcntl function is useful in changing the access control flag of a file descriptor.
- ✓ For example: after a file is opened for blocking read-write access and the process needs to change the access to non-blocking and in write-append mode, it can call:

```
int cur_flags=fcntl(fdesc,F_GETFL);
int rc=fcntl(fdesc,F_SETFL,cur_flag | O_APPEND | O_NONBLOCK);
```

The following example reports the close-on-exec flag of fdesc, sets it to on afterwards:

```
cout<<fdesc<<”close-on-exec”<<fcntl(fdesc,F_GETFD)<<endl;
(void)fcntl(fdesc,F_SETFD,1); //turn on close-on-exec flag
```

The following statements change the standard input og a process to a file called FOO:

```
int fdesc=open(“FOO”,O_RDONLY);   //open FOO for read
close(0);                                 //close standard input
if(fcntl(fdesc,F_DUPFD,0)==-1)
      perror(“fcntl”);                //stdin from FOO now
char buf[256];
int rc=read(0,buf,256);               //read data from FOO
```

The dup and dup2 functions in UNIX perform the same file duplication function as fcntl.
They can be implemented using fcntl as:

```
#define dup(fdesc)                  fcntl(fdesc, F_DUPFD,0)
#define dup2(fdesc1,fd2)            close(fd2),fcntl(fdesc,F_DUPFD,fd2)
```

lseek

- The lseek function is also used to change the file offset to a different value.
- Thus lseek allows a process to perform random access of data on any opened file.
- The prototype of lseek is

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fdesc, off_t pos, int whence);
```

- On success it returns new file offset, and –1 on error.
- The first argument fdesc, is an integer file descriptor that refer to an opened file.
- The second argument pos, specifies a byte offset to be added to a reference location in deriving the new file offset value.
- The third argument whence, is the reference location.

Whence value	Reference location
SEEK_CUR	Current file pointer address
SEEK_SET	The beginning of a file
SEEK_END	The end of a file

- They are defined in the <unistd.h> header.
- If an lseek call will result in a new file offset that is beyond the current end-of-file, two outcomes possible are:
    - If a file is opened for read-only, lseek will fail.
    - If a file is opened for write access, lseek will succeed.
    - The data between the end-of-file and the new file offset address will be initialised with NULL characters.

link

- The link function creates a new link for the existing file.
- The prototype of the link function is

```
#include <unistd.h>
int link(const char *cur_link, const char *new_link);
```

- If successful, the link function returns 0.
- If unsuccessful, link returns –1.
- The first argument cur_link, is the pathname of existing file.
- The second argument new_link is a new pathname to be assigned to the same file.
- If this call succeeds, the hard link count will be increased by 1.
- The UNIX ln command is implemented using the link API.

```
    /*test_ln.c*/
    #include<iostream.h>
    #include<stdio.h>
    #include<unistd.h>

    int main(int argc, char* argv)
    {
        if(argc!=3)
        {
              cerr<<”usage:”<<argv[0]<<”<src_file><dest_file>\n”;
              return 0;
        }
        if(link(argv[1],argv[2])==-1)
        {
              perror(“link”);
              return 1;
        }
        return 0;
}
```

unlink

- The unlink function deletes a link of an existing file.
- This function decreases the hard link count attributes of the named file, and removes the file name entry of the link from directory file.
- A file is removed from the file system when its hard link count is zero and no process has any file descriptor referencing that file.
- The prototype of unlink is

```
#include <unistd.h>
int unlink(const char * cur_link);
```

- If successful, the unlink function returns 0.
- If unsuccessful, unlink returns –1.
- The argument cur_link is a path name that references an existing file.
- ANSI C defines the rename function which does the similar unlink operation.
- The prototype of the rename function is:

```
#include<stdio.h>
int rename(const char * old_path_name,const char * new_path_name);
```

- The UNIX mv command can be implemented using the link and unlink APIs as shown:

```
#include <iostream.h>
#include <unistd.h>
#include<string.h>
int main ( int argc, char *argv[ ])
{
      if (argc != 3 || strcmp(argv[1],argcv[2]))
            cerr<<”usage:”<<argv[0]<<””<old_link><new_link>\n”;
      else if(link(argv[1],argv[2]) == 0)
            return unlink(argv[1]);
      return 1;
}
```

stat, fstat

- The stat and fstat function retrieves the file attributes of a given file.
- The only difference between stat and fstat is that the first argument of a stat is a file pathname, where as the first argument of fstat is file descriptor.
- The prototypes of these functions are

```
#include<sys/stat.h>
#include<unistd.h>

int stat(const char *pathname, struct stat *statv);
int fstat(const int fdesc, struct stat *statv);
```

- ✓ The second argument to stat and fstat is the address of a struct stat-typed variable which is defined in the <sys/stat.h> header.
- ✓ Its declaration is as follows:

```
struct stat
{
dev_t       st_dev;      /* file system ID */
ino_t    st_ino;         /* file inode number */
mode_t      st_mode;   /* contains file type and permission */
nlink_t  st_nlink;  /* hard link count */
uid_t       st_uid;      /* file user ID */
gid_t       st_gid;      /* file group ID */
dev_t       st_rdev;     /*contains major and minor device#*/
off_t       st_size;   /* file size in bytes */
time_t      st_atime;  /* last access time */
time_t      st_mtime;  /* last modification time */
time_t      st_ctime;   /* last status change time */
};
```

- ✓ The return value of both functions is
  - ○ 0 if they succeed
  - ○ -1 if they fail
  - ○ *errno* contains an error status code
- ✓ The lstat function prototype is the same as that of stat:

```
int lstat(const char * path_name, struct stat* statv);
```

- ✓ We can determine the file type with the macros as shown.

macro	**Type of file**
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

Note: refer UNIX lab program 3(b) for example

❖ access

- ✓ The access system call checks the existence and access permission of user to a named file.
- ✓ The prototype of access function is:

```
#include<unistd.h>
int access(const char *path_name, int flag);
```

- ✓ On success access returns 0, on failure it returns –1.
- ✓ The first argument is the pathname of a file.
- ✓ The second argument flag, contains one or more of the following bit flag .

Bit flag	**Uses**
F_OK	Checks whether a named file exist
R_OK	Test for read permission
W_OK	Test for write permission
X_OK	Test for execute permission

- ✓ The flag argument value to an access call is composed by bitwise-ORing one or more of the above bit flags as shown:

```
int rc=access("/usr/divya/usp.txt",R_OK | W_OK);
```

- ✓ example to check whether a file exists:

```
if(access("/usr/divya/usp.txt", F_OK)==-1)
       printf("file does not exists");
else
       printf("file exists");
```

chmod, fchmod

- The chmod and fchmod functions change file access permissions for owner, group & others as well as the set_UID, set_GID and sticky flags.
- A process must have the effective UID of either the super-user/owner of the file.
- The prototypes of these functions are

```
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>

int chmod(const char *pathname, mode_t flag);
int fchmod(int fdesc, mode_t flag);
```

- The pathname argument of chmod is the path name of a file whereas the fdesc argument of fchmod is the file descriptor of a file.
- The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened.
- To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have super-user permissions. The mode is specified as the bitwise OR of the constants shown below.

Mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

chown, fchown, lchown

- The chown functions changes the user ID and group ID of files.
- The prototypes of these functions are

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

int chown(const char *path_name, uid_t uid, gid_t gid);
int fchown(int fdesc, uid_t uid, gid_t gid);
int lchown(const char *path_name, uid_t uid, gid_t gid);
```

- The path_name argument is the path name of a file.
- The uid argument specifies the new user ID to be assigned to the file.
- The gid argument specifies the new group ID to be assigned to the file.

```
/* Program to illustrate chown function */
#include<iostream.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
#include<pwd.h>
```

```
int main(int argc, char *argv[ ])
{
        if(argc>3)
        {
                cerr<<”usage:”<<argv[0]<<”<usr_name><file>....\n”;
                return 1;
        }

        struct passwd *pwd = getpwuid(argv[1]) ;
        uid_t           UID = pwd ? pwd -> pw_uid : -1 ;
        struct          stat   statv;

        if (UID == (uid_t)-1)
                cerr <<“Invalid user name”;
        else for (int i = 2; i < argc ; i++)
                if (stat(argv[i], &statv)==0)
                {
                        if (chown(argv[i], UID,statv.st_gid))
                                perror (“chown”);
                        else
                                perror (“stat”);
                }
        return 0;
}
```

- ✓ The above program takes at least two command line arguments:
  - ○ The first one is the user name to be assigned to files
  - ○ The second and any subsequent arguments are file path names.
- ✓ The program first converts a given user name to a user ID via *getpwuid* function. If that succeeds, the program processes each named file as follows: it calls *stat* to get the file group ID, then it calls *chown* to change the file user ID. If either the stat or chown fails, error is displayed.

❖ utime Function

- ✓ The utime function modifies the access time and the modification time stamps of a file.
- ✓ The prototype of utime function is

```
#include<sys/types.h>
#include<unistd.h>
#include<utime.h>

int utime(const char *path_name, struct utimbuf *times);
```

- ✓ On success it returns 0, on failure it returns –1.
- ✓ The path_name argument specifies the path name of a file.
- ✓ The times argument specifies the new access time and modification time for the file.
- ✓ The struct utimbuf is defined in the <utime.h> header as:

```
struct utimbuf
{
        time_t          actime;          /* access time */
        time_t          modtime;         /* modification time */
}
```

- ✓ The time_t datatype is an unsigned long and its data is the number of the seconds elapsed since the birthday of UNIX : 12 AM , Jan 1 of 1970.
- ✓ If the times (variable) is specified as NULL, the function will set the named file access and modification time to the current time.
- ✓ If the times (variable) is an address of the variable of the type struct utimbuf, the function will set the file access time and modification time to the value specified by the variable.

File and Record Locking

- Multiple processes performs read and write operation on the same file concurrently.
- This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file.

- So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism.
- File locking is applicable for regular files.
- Only a process can impose a write lock or read lock on either a portion of a file or on the entire file.
- The differences between the read lock and the write lock is that when write lock is set, it prevents the other process from setting any over-lapping read or write lock on the locked file.
- Similarly when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region.
- The intension of the write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region, so write lock is termed as "**Exclusive lock**".
- The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region.
- Other processes are allowed to lock and read data from the locked regions. Hence a read lock is also called as "**shared lock** ".
- File lock may be **mandatory** if they are enforced by an operating system kernel.
- If a mandatory exclusive lock is set on a file, no process can use the read or write system calls to access the data on the locked region.
- These mechanisms can be used to synchronize reading and writing of shared files by multiple processes.
- If a process locks up a file, other processes that attempt to write to the locked regions are blocked until the former process releases its lock.
- Problem with mandatory lock is – if a runaway process sets a mandatory exclusive lock on a file and never unlocks it, then, no other process can access the locked region of the file until the runway process is killed or the system has to be rebooted.
- If locks are not mandatory, then it has to be **advisory** lock.
- A kernel at the system call level does not enforce advisory locks.
- This means that even though a lock may be set on a file, no other processes can still use the read and write functions to access the file.
- To make use of advisory locks, process that manipulate the same file must co-operate such that they follow the given below procedure for every read or write operation to the file.
    1. Try to set a lock at the region to be accesses. If this fails, a process can either wait for the lock request to become successful.
    2. After a lock is acquired successfully, read or write the locked region.
    3. Release the lock.
- If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512, the previous read lock from 0 to 256 is now covered by the write lock and the process does not own two locks on the region from 0 to 256. This process is called "**Lock Promotion**".
- Furthermore, if a process now unblocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called "**Lock Splitting**".
- UNIX systems provide fcntl function to support file locking. By using fcntl it is possible to impose read or write locks on either a region or an entire file.
- The prototype of fcntl is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd_flag, ....);
```

- The first argument specifies the file descriptor.
- The second argument cmd_flag specifies what operation has to be performed.
- If fcntl is used for file locking then it can values as

F_SETLK	sets a file lock, do not block if this cannot succeed immediately.
F_SETLKW	sets a file lock and blocks the process until the lock is acquired.
F_GETLK	queries as to which process locked a specified region of file.

- For file locking purpose, the third argument to fctnl is an address of a *struct flock* type variable.
- This variable specifies a region of a file where lock is to be set, unset or queried.

```
struct flock
{
```

```
     short  l_type;   /* what lock to be set or to unlock file */
     short  l_whence;   /* Reference address for the next field */
     off_t  l_start ;   /*offset from the l_whence reference addr*/
     off_t  l_len ;   /*how many bytes in the locked region */
     pid_t  l_pid ;    /*pid of a process which has locked the file */
     };
```

- The l_type field specifies the lock type to be set or unset.
- The possible values, which are defined in the <fcntl.h> header, and their uses are

l_type value	**Use**
F_RDLCK	Set a read lock on a specified region
F_WRLCK	Set a write lock on a specified region
F_UNLCK	Unlock a specified region

- The l_whence, l_start & l_len define a region of a file to be locked or unlocked.
- The possible values of l_whence and their uses are

l_whence value	**Use**
SEEK_CUR	The l_start value is added to current file pointer address
SEEK_SET	The l_start value is added to byte 0 of the file
SEEK_END	The l_start value is added to the end of the file

- A lock set by the fcntl API is an advisory lock but we can also use fcntl for mandatory locking purpose with the following attributes set before using fcntl
  1. Turn on the set-GID flag of the file.
  2. Turn off the group execute right permission of the file.
- In the given example program we have performed a read lock on a file “divya” from the 10th byte to 25th byte.

Example Program

```
#include <unistd.h>
#include<fcntl.h>
int main ( )
{
       int fd;
       struct flock lock;
       fd=open(“divya”,O_RDONLY);
       lock.l_type=F_RDLCK;
       lock.l_whence=0;
       lock.l_start=10;
       lock.l_len=15;
       fcntl(fd,F_SETLK,&lock);
}
```

Directory Fil e API’s

- A Directory file is a record-oriented file, where each record stores a file name and the inode number of a file that resides in that directory.
- Directories are created with the mkdir API and deleted with the rmdir API.
- The prototype of mkdir is

```
#include<sys/stat.h>
#include<unistd.h>

int mkdir(const char *path_name, mode_t mode);
```

- The first argument is the path name of a directory file to be created.
- The second argument mode, specifies the access permission for the owner, groups and others to be assigned to the file. This function creates a new empty directory.
- The entries for “.” and “..” are automatically created. The specified file access permission, mode, are modified by the file mode creation mask of the process.

- To allow a process to scan directories in a file system independent manner, a directory record is defined as ***struct dirent*** in the <dirent.h> header for UNIX.
- Some of the functions that are defined for directory file operations in the above header are

```
#include<sys/types.h>
#if defined (BSD)&&!_POSIX_SOURCE
       #include<sys/dir.h>
       typedef struct direct Dirent;
#else
       #include<dirent.h>
       typedef struct direct Dirent;
#endif
```

```
 DIR *opendir(const char *path_name);
 Dirent *readdir(DIR *dir_fdesc);
 int closedir(DIR *dir_fdesc);
void rewinddir(DIR *dir_fdsec);
```

The uses of these functions are

Function	Use
opendir	Opens a directory file for read-only. Returns a file handle dir * for future reference of the file.
readdir	Reads a record from a directory file referenced by dir-fdesc and returns that record information.
rewinddir	Resets the file pointer to the beginning of the directory file referenced by dir-fdesc. The next call to readdir will read the first record from the file.
closedir	closes a directory file referenced by dir-fdesc.

- An empty directory is deleted with the rmdir API.
- The prototype of rmdir is

```
#include<unistd.h>
int rmdir (const char * path_name);
```

- If the link count of the directory becomes 0, with the call and no other process has the directory open then the space occupied by the directory is freed.
- UNIX systems have defined additional functions for random access of directory file records.

Function	Use
telldir	Returns the file pointer of a given dir_fdesc
seekdir	Changes the file pointer of a given dir_fdesc to a specified address

UNIX PROCESSES

INTRODUCTION

A Process is a program under execution in a UNIX or POSIX system.

UNIX shell is a process that is created when a user logs on to a system.

When user enters a command shell creates a new process. This is a child process.

The advantages of allowing any process to create new processes in its course of execution are:

1. Any user can create multitasking applications.
2. Because a child process executes in its own virtual address space, its success or failure in execution will not affect its parent.
3. It is very common for a process to create a child process that will execute a new program. This allows users to write programs that can call on any other program to extend their functionality without the need to incorporate any new source code.

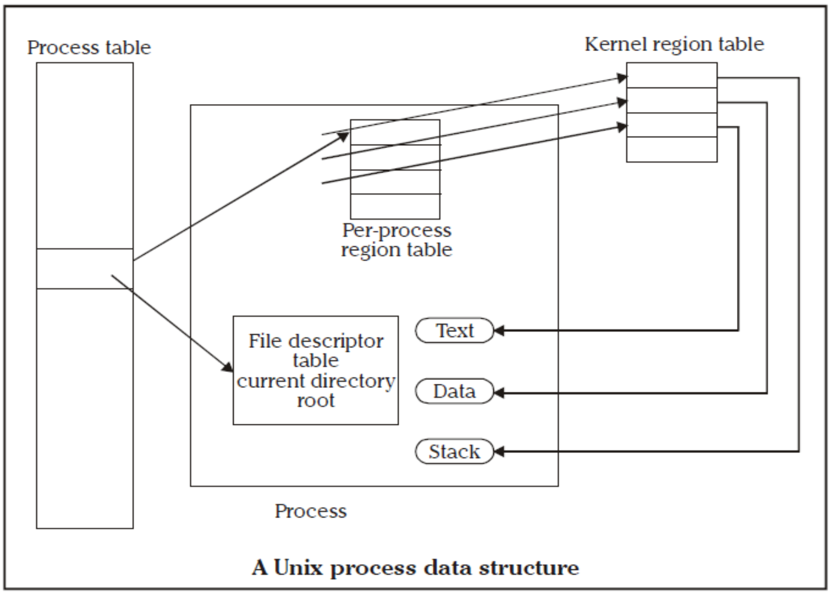
UNIX KERNEL SUPPORT FOR PROCESS

The data structure and execution of processes are dependent on operating system implementation.

A UNIX process consists minimally of a text segment, a data segment and a stack segment. A segment is an area of memory that is managed by the system as a unit.

- A text segment consists of the program text in machine executable instruction code format.
- The data segment contains static and global variables and their corresponding data.
- A stack segment contains runtime variables and the return addresses of all active functions for a process. UNIX kernel has a process table that keeps track of all active process present in the system. Some of these processes

~~belongs to the kernel and are called as "system process". Every entry in the process table contains pointers to the~~ text, data and the stack segments and also to U-area of a process. U-area of a process is an extension of the process table entry and contains other process specific data such as the file descriptor table, current root and working directory inode numbers and set of system imposed process limits.



A Unix process data structure

All processes in UNIX system expect the process that is created by the system boot code, are created by the fork system call. After the fork system call, once the child process is created, both the parent and child processes resumes execution. When a process is created by fork, it contains duplicated copies of the text, data and stack segments of its parent as shown in the Figure below. Also it has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

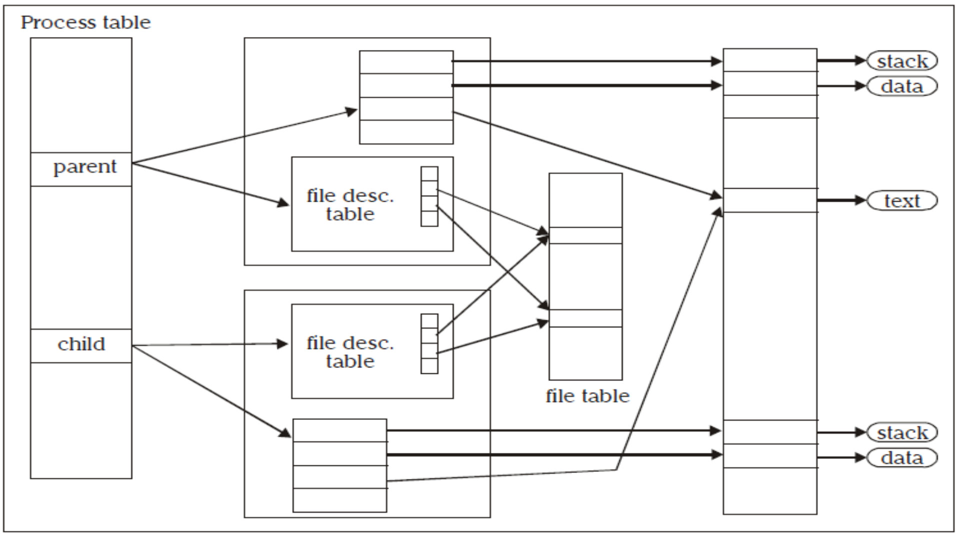


Figure: Parent & child relationship after fork

The process will be assigned with attributes, which are either inherited from its parent or will be set by the kernel.

- **A real user identification number (rUID)**: the user ID of a user who created the parent process.
- **A real group identification number (rGID)**: the group ID of a user who created that parent process.
- **An effective user identification number (eUID)**: this allows the process to access and create files with the same privileges as the program file owner.
- **An effective group identification number (eGID)**: this allows the process to access and create files with the same privileges as the group to which the program file belongs.
- **Saved set-UID and saved set-GID**: these are the assigned eUID and eGID of the process respectively.
- **Process group identification number (PGID) and session identification number (SID):** these identify the

process group and session of which the process is member.

- **Supplementary group identification numbers:** this is a set of additional group IDs for a user who created the process.
- **Current directory:** this is the reference (inode number) to a working directory file.
- **Root directory**: this is the reference to a root directory.
- **Signal handling**: the signal handling settings.
- **Signal mask**: a signal mask that specifies which signals are to be blocked.
- **Unmask**: a file mode mask that is used in creation of files to specify which accession rights should be taken out.
- **Nice value**: the process scheduling priority value.
- **Controlling terminal**: the controlling terminal of the process.

In addition to the above attributes, the following attributes are different between the parent and child processes:

- **Process identification number (PID)**: an integer identification number that is unique per process in an entire operating system.
- **Parent process identification number (PPID)**: the parent process PID.
- **Pending signals**: the set of signals that are pending delivery to the parent process.
- **Alarm clock time**: the process alarm clock time is reset to zero in the child process.
- **File locks**: the set of file locks owned by the parent process is not inherited by the chid process.

fork and ***exec*** are commonly used together to spawn a sub-process to execute a different program. The advantages of this method are:

- A process can create multiple processes to execute multiple programs concurrently.
- Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of its child process.

PROCESS CONTROL

INTRODUCTION

Process control is concerned about creation of new processes, program execution, and process termination.

PROCESS Attributes

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

```
pid_t getpid(void);
```

Returns: process ID of calling process

```
pid_t getppid(void);
```

Returns: parent process ID of calling process

```
pid_t getpgrp(void);
```

Returns: group process ID of calling process

```
uid_t getuid(void);
```

Returns: real user ID of calling process

```
uid_t geteuid(void);
```

Returns: effective user ID of calling process

```
gid_t getgid(void);
```

Returns: real group ID of calling process

```
gid_t getegid(void);
```

Returns: effective group ID of calling process

- ✓ Getpgrp : When a user logs onto a system, the login process becomes a session leader and process group leader. The session ID and the process group ID of a session leader is the same as its process ID.
  - o If the login process creates new child processes to execute jobs, these child processes will be in the same session and process group as the login process.

fork FUNCTION

An existing process can create a new one by calling the `fork` function.

```
#include <unistd.h>
pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, 1 on error.

- The new process created by `fork` is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason `fork` returns 0 to the child is that a process can have only a single parent, and the child can always call `getppid` to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)
- Both the child and the parent continue executing with the instruction that follows the call to `fork`.
- The child is a copy of the parent.
- For example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; the parent and the child do not share these portions of memory.
- The parent and the child share the text segment .

Example programs:

Program 1

/* Program to demonstrate fork function Program name – fork1.c */

```
#include<sys/types.h>
#include<unistd.h>
int main( )
{
	fork( );
	printf("\n hello USP");
}
```

Output :

```
$ cc fork1.c
$  ./a.out
hello  USP
hello USP
```

Note : The statement hello USP is executed twice as both the child and parent have executed that instruction.

Program 2

```
/* Program name – fork2.c */
#include<sys/types.h>
#include<unistd.h>
int main( )
{
        printf(“\n 6 sem “);
        fork( );
        printf(“\n hello USP”);
}
```

Output :

```
$ cc fork1.c
$ ./a.out
6 sem
hello USP
hello USP
```

Note: The statement 6 sem is executed only once by the parent because it is called before fork and statement hello USP is executed twice by child and parent.

There are numerous other properties of the parent that are inherited by the child:

- Real user ID, real group ID, effective user ID, effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

The differences between the parent and child are

- The return value from `fork`
- The process IDs are different
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change
- The child's `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` values are set to 0
- File locks set by the parent are not inherited by the child
- Pending alarms are cleared for the child
- The set of pending signals for the child is set to the empty set

The two main reasons for `fork` to fail are

(a)ENOMEM if too many processes are already in the system, which usually means that insufficient memory to create the new process (b)EAGAIN if the total number of processes for this real user ID exceeds the system's limit.

CHILD_MAX	Maximum no. of processes that can be created by a single user.	,<sys/param.h>
MAXPID	Maximum no. of processes that can exist concurrently system-wide	<limits.h>

Both the child and parent process will be scheduled by the UNIX kernel to run independently and order of execution is implementation dependent.

There are two uses for fork:

- ❖ When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. This is common for network servers, the parent waits for a service request from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
- ❖ When a process wants to execute a different program. This is common for shells. In this case, the child does an exec right after it returns from the fork.

vfork FUNCTION

- ✓ The function vfork has the same calling sequence and same return values as fork.
- ✓ The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.
- ✓ The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls exec (or exit) right after the vfork.
- ✓ Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System.
- ✓ Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes.
- ✓ Vfork is unsafe to use because if the child process modifies any data of the parent before it calls exec or _exit those changes will remain when the parent process resumes execution.
- ✓ Vfork should be used in porting old applications to the new UNIX systems only.
- ✓ Copy on write : the latest UNIX systems have improved on the efficiency of fork by allowing parent and child processes to share common virtual address space until the child calls either the exec or _exit function.

  If parent or child modifies any data the kernel creates new memory pages that cover virtual address space modified. The process that has made change s will reference new memory pages and other process will refer old memory pages. This process is called copy on write.

wait AND waitpid FUNCTIONS

These are used by parent process to wait for its child process to terminate and to retrieve the child exit status.

These calls will deallocate the process table slot of the child process, so that slot can be reused by a new process.

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event - it can happen at any time while the parent is running - this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.

A process that calls `wait` or `waitpid` can:

- ✓ Block, if all of its children are still running
- ✓ Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- ✓ Return immediately with an error, if it doesn't have any child processes.

#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);

Both return: process ID if OK, 0 (see later), or -1 on error.

The differences between these two functions are as follows.

- The `wait` function can block the caller until a child process terminates, where as `waitpid` has an option that prevents it from blocking.
- The `waitpid` function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, `wait` returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates.

If the caller blocks and has multiple children, `wait` returns when one terminates.

For both functions, the argument statloc(status_p) is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

If this argument is a null no child exit status to be queried.

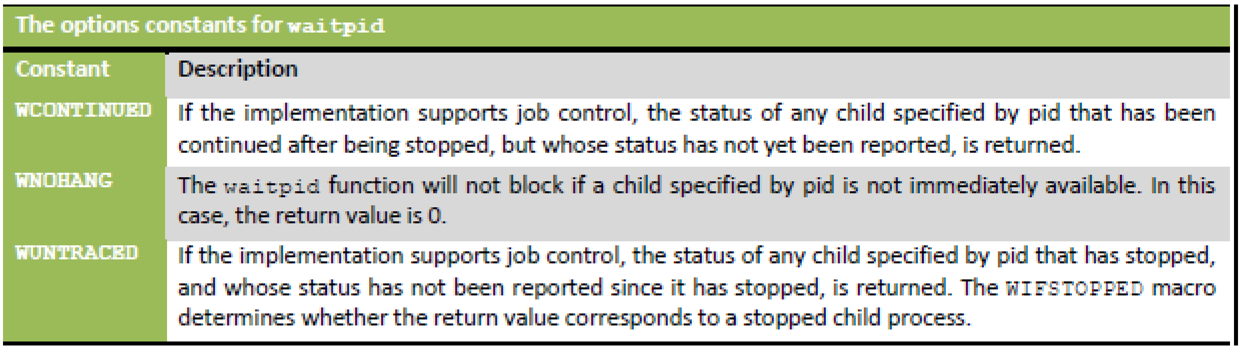
The interpretation of the pid argument for `waitpid` depends on its value:

pid == 1	Waits for any child process. In this respect, waitpid is equivalent to wait.
pid > 0	Waits for the child whose process ID equals pid.
pid == 0	Waits for any child whose process group ID equals that of the calling process.
pid < 1	Waits for any child whose process group ID equals the absolute value of pid.

Parent can check the exit status code with the following macros as defined in <sys/wait.h>

WIFEXITED(*STATUS_P)	Returns a nonzero value if child was terminated via an _exit call and zero otherwise
WEXITSTATUS(*STATUS_P)	Returns a child exit status code that was assigned to _exit call. This should be called only if WIFEXITED returns a nonzero value.
WIFSIGNALED(*STATUS_P)	Returns a nonzero value if child was terminated due to

	signal interruption
WTERMSIG(*STATUS_P)	Returns a signal number that had terminated a child process. This should be called only if WIFSIGNALED returns a nonzero value
WIFSTOPPED(*STATUS_P)	Returns a nonzero value if child has stopped due to job control.
WSTOPSIG(*STATUS_P)	Returns a signal number that has stopped child process. This should be called only if WIFSTOPPED returns a nonzero value



The options constants for `waitpid`

Constant	Description
`WCONTINUED`	If the implementation supports job control, the status of any child specified by pid that has been continued after being stopped, but whose status has not yet been reported, is returned.
`WNOHANG`	The `waitpid` function will not block if a child specified by pid is not immediately available. In this case, the return value is 0.
`WUNTRACED`	If the implementation supports job control, the status of any child specified by pid that has stopped, and whose status has not been reported since it has stopped, is returned. The `WIFSTOPPED` macro determines whether the return value corresponds to a stopped child process.

RERURN VALUE :

Positive integer : child PID
-1 : no child satisfied the wait criteria or the function was interrupted by a caught signal.

Errno has following values :

EINTR	Wait or waitpid returns because the system call was interrupted by signal
ECHILD	Wait : calling process has no unwaited for child process Waitpid : either the child_pid value is illegal or process can't be in a state as defined by the options value.
EFAULT	Status_p points to an illegal address
EINVAL	Option value is illegal

The `waitpid` function provides three features that aren't provided by the `wait` function.

- ✓ The `waitpid` function lets us wait for one particular process, whereas the `wait` function returns the status of any terminated child.

- ✓ The `waitpid` function provides a nonblocking version of `wait`. There are times when we want to fetch a child's status, but we don't want to block.
- ✓ The `waitpid` function provides support for job control with the `WUNTRACED` and `WCONTINUED` options.

`exec` FUNCTIONS

When a process calls one of the `exec` functions, that process is completely replaced by the new program, and the new program starts executing at its `main` function. The process ID does not change across an `exec`, because a new process is not created; `exec` merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk.

There are 6 exec functions:

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0,... /* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int execle(const char *pathname, const char *arg0,... /*(char *)0, char
*const envp */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv []);
```

All six return: -1 on error, no return on success.

- ❖ The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument. When a filename argument is specified
  - If filename contains a slash, it is taken as a pathname.
  - Otherwise, the executable file is searched for in the directories specified by the `PATH` environment variable.
- ❖ The next difference concerns the passing of the argument list (`l` stands for list and `v` stands for vector). The functions `execl`, `execlp`, and `execle` require each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (`execv`, `execvp`, and `execve`), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.
- ❖ The final difference is the passing of the environment list to the new program. The two functions whose names end in an `e` (`execle` and `execve`) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the `environ` variable in the calling process to copy the existing environment for the new program.

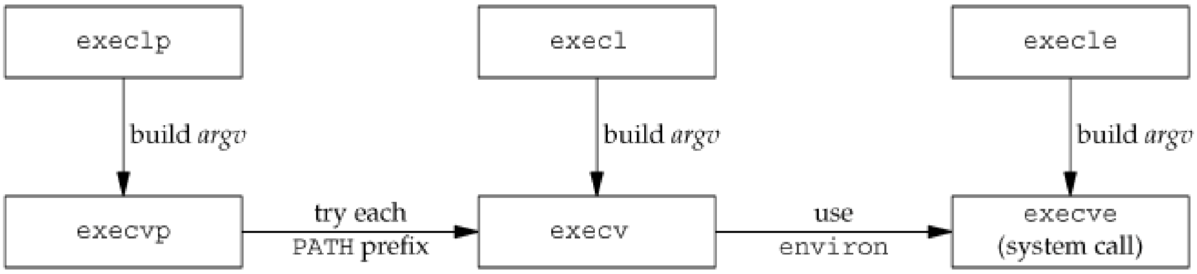
Function	pathname	filename	Arg list	argv[]	`environ`	envp[]
`execl`	•		•		•	
`execlp`		•	•		•	
`execle`	•		•			•
`execv`	•			•	•	
`execvp`		•		•	•	
`execve`	•			•		•
(letter in name)		`p`	`l`	`v`		`e`

The above table shows the differences among the 6 exec functions.

We've mentioned that the process ID does not change after an `exec`, but the new program inherits additional properties from the calling process:

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal

- Time left until alarm clock
- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Values for `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`.



1. The three functions in the top row specify each argument string as a separate argument to the exec function, with a null pointer terminating the variable number of arguments. The three functions in the second row have an *argv* array, containing pointers to the argument strings. This *argv* array must contain a null pointer to specify its end, since a count is not specified.

2. The two functions in the left column specify a *filename* argument. This is converted into a *pathname* using the current PATH environment variable. If the *filename* argument to execlp or execvp contains a slash (/) anywhere in the string, the PATH variable is not used. The four functions in the right two columns specify a fully qualified *pathname* argument.

3. The four functions in the left two columns do not specify an explicit environment pointer. Instead, the current value of the external variable environ is used for building an environment list that is passed to the new program. The two functions in the right column specify an explicit environment list. The *envp* array of pointers must be terminated by a null pointer.

PROCESS TERMINATION

There are eight ways for a process to terminate.
Normal termination occurs in five ways:

- Return from `main`
- Calling `exit`
- Calling `_exit` or `_Exit`
- Return of the last thread from its start routine
- Calling `pthread_exit` from the last thread

Abnormal termination occurs in three ways:

- Calling `abort`
- Receipt of a signal
- Response of the last thread to a cancellation request

exit FUNCTIONS

A process can terminate normally in five ways:

- Executing a return from the main function.
- Calling the exit function.

- Calling the _exit or _Exit function.
  In most UNIX system implementations, exit(3) is a function in the standard C library, whereas _exit(2) is a system call.
- Executing a return from the start routine of the last thread in the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
- Calling the pthread_exit function from the last thread in the process.

The three forms of abnormal termination are as follows:

- Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.
- When the process receives certain signals. Examples of signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.
- The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later, the target thread terminates.

Exit Functions

Three functions terminate a program normally: `_exit` and `_Exit`, which return to the kernel immediately, and `exit`, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>
void exit(int status);
void _Exit(int status);

#include <unistd.h>
void _exit(int status);
```

All three exit functions expect a single integer argument, called the exit status. Returning an integer value from the main function is equivalent to calling exit with the same value. Thus

```
exit(0);
```

is the same as

```
return(0);
```

from the main function.

In the following situations the exit status of the process is undefined.

✓ any of these functions is called without an exit status.
✓ main does a return without a return value
✓ main “falls off the end”, i.e if the exit status of the process is undefined.

Example:

```
$ cc hello.c
    $ ./a.out
    hello, world
    $ echo $?                     //  print the exit status
    13
```

`atexit` Function

With ISO C, a process can register up to 32 functions that are automatically called by `exit`. These are called exit handlers and are registered by calling the `atexit` function.

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Returns: 0 if OK, nonzero on error

This declaration says that we pass the address of a function as the argument to `atexit`. When this function is called, it is not passed any arguments and is not expected to return a value. The `exit` function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

Example of exit handlers

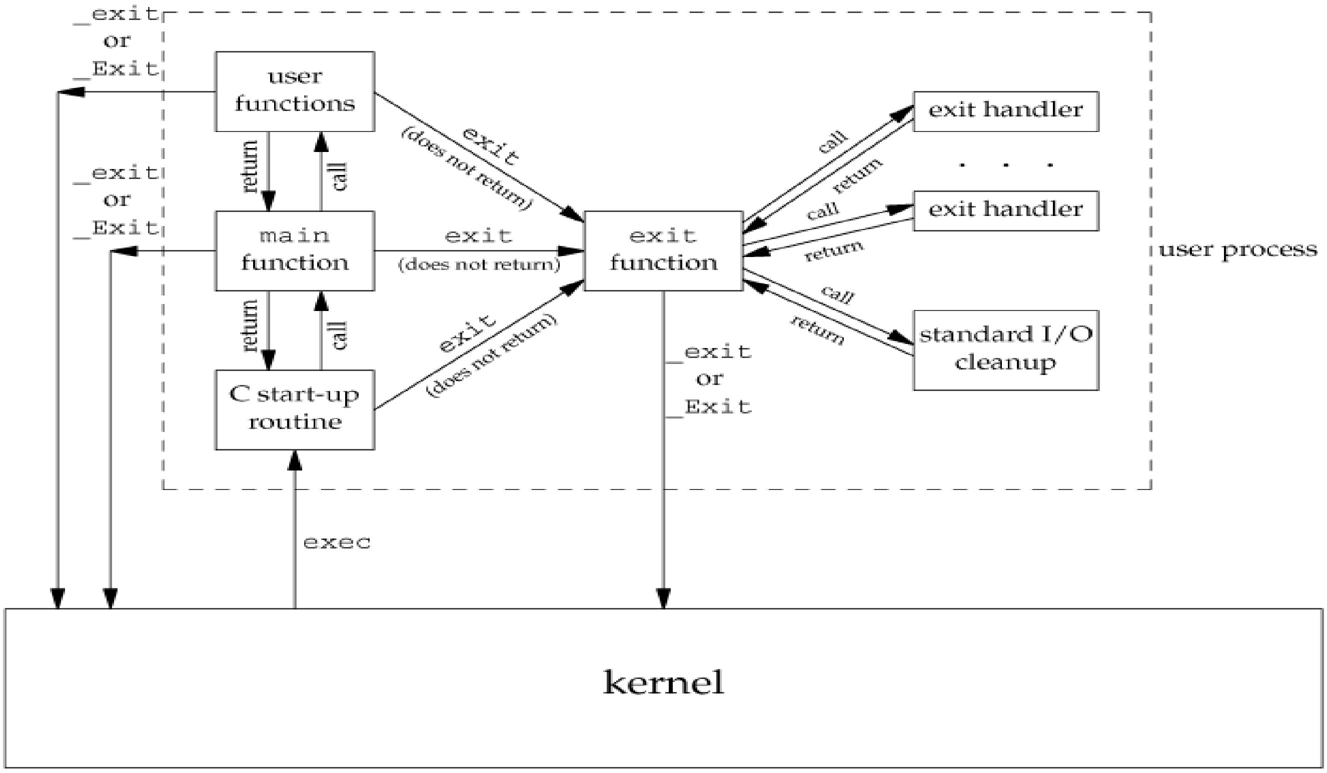
```
#include "apue.h"
```

```
static void my_exit1(void);
static void my_exit2(void);

int main(void)
{
     if (atexit(my_exit2) != 0)
         err_sys("can't register my_exit2");

     if (atexit(my_exit1) != 0)
         err_sys("can't register my_exit1");

     if (atexit(my_exit1) != 0)
         err_sys("can't register my_exit1");

     printf("main is done\n");
     return(0);
}

static void
my_exit1(void)
{
   printf("first exit handler\n");
}

static void
my_exit2(void)
{
   printf("second exit handler\n");
}
```

Output:

```
$ ./a.out
    main is done
    first exit handler
    first exit handler
    second exit handler
```

The below figure summarizes how a C program is started and the various ways it can terminate.



❖ system Function:

- It is convenient to execute a command string from within a program.

  #include <stdlib.h>
  int *system*(const char *cmdstring);
- If cmdstring is a null pointer, *system* returns nonzero only if a command processor is available. This feature determines whether the *system* function is supported on a given operating system. Under the UNIX System, *system* is always available.
- The *system* is implemented by calling fork, exec, and waitpid, there are three types of return values.
- If either the fork fails or waitpid returns an error ,system returns 1 with errno set to indicate the error.
- If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).
- Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from *system* is the termination status of the shell, in the format specified for waitpid.

❖ Zombie process :

Process whose parents don’t wait for their death move to zombie state.

When the process dies, its parent picks up the child’s exit status (the reason for waiting) from the process table and frees up the process table entry. However, when the parent doesn’t wait (but is still alive), the child turns into a zombie. A zombie is a harmless dead child that reserves the process table slot.

The parent is sent a SIGCHLD signal indicating that a child has died; the handler for this signal will typically execute the wait system call, which reads the exit status and removes the zombie. The zombie's process ID and entry in the process table can then be reused. However, if a parent ignores the SIGCHLD, the zombie will be left in the process table.

- To avoid creating zombie processes, the parent must:
    1. Handle the SIGCHLD signal.
    2. In this signal handling function, the parent must wait for the child process.

```
$ ps
        PID   TTY        TIME    CMD
        1074   pts/2   00:00:00   bash
        1280   pts/2   00:00:00   parentTest.exe
        1281   pts/2   00:00:00   childTest.exe <defunct>
        1288   pts/2   00:00:00    ps
```

```
$ ps -l
  F    S   UID   PID  PPID  C PRI  NI ADDR    SZ  WCHAN TTY         TIME CMD
000 S   561    1074  1073  0  76   0    -   628 11a418 pts/2    00:00:00 bash
000 S   561    1301  1074  0  70   0    -   436 11f22b pts/2    00:00:00 parentTes
004 Z   561   1302  1301  0  70   0    -     0 119ffb pts/2    00:00:00 childTest
000 R   561   1320  1074  0  77   0    -   646      - pts/2    00:00:00 ps
```

❖ Orphan process :

- A process whose parent has exited.
- Orphaned processes are inherited by *init*
- Its slot in the process table is immediately released when an orphan terminates.

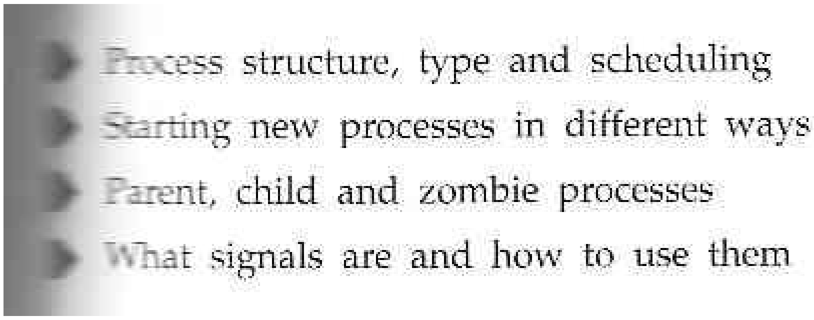
Unit II

Processes and Signals

III. Processes and Signals

Processes and signals form a fundamental part of the UNIX operating environment, controlling almost all activities performed by a UNIX computer system.

Here are some of the things you need to understand.



- Process structure, type and scheduling
- Starting new processes in different ways
- Parent, child and zombie processes
- What signals are and how to use them

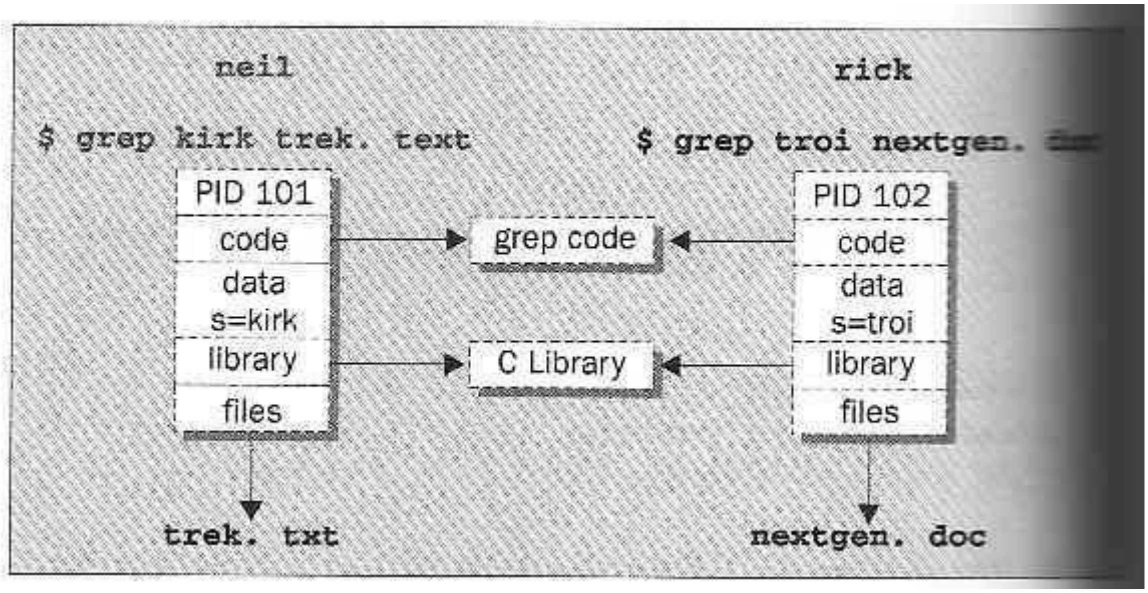
3.1 What is a Process

The X/Open Specification defines a process as an address space and single thread of control that executes within that address space and its required system resources.

A process is, essentially, a running program.

3. 2 Layout of a C program

Here is how a couple of processes might be arranged within the operationg system.



Each process is allocated a unique number, a **process identifier**, or PID.

The program code that will be executed by the **grep** command is stored in a disk file.

The system libraries can also be shared.

A process has its own stack space.

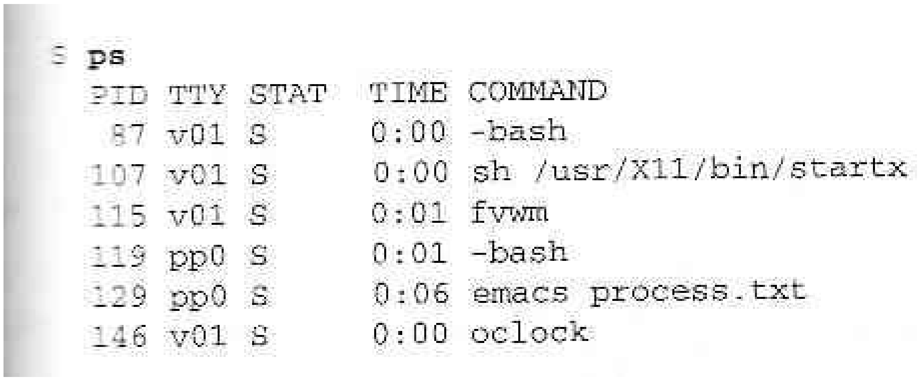
Image in main memory

The UNIX **process table** may be though of as a data structure describing all of the processes that are currently loaded.

Viewing Processes

We can see what processes are running by using the **ps** command.

Here is some sample output:



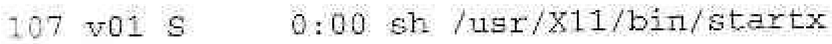
```
$ ps
  PID TTY STAT  TIME COMMAND
   87 v01 S     0:00 -bash
  107 v01 S     0:00 sh /usr/X11/bin/startx
  115 v01 S     0:01 fvwm
  119 pp0 S     0:01 -bash
  129 pp0 S     0:06 emacs process.txt
  146 v01 S     0:00 oclock
```

The **PID** column gives the PIDs, the **TTY** column shows which terminal started the process, the **STAT** column shows the current status, **TIME** gives the CPU time used so far and the **COMMAND** column shows the command used to start the process.

Let's take a closer look at some of these:

```
87 v01 S     0:00 -bash
```

The initial login was performed on virtual console number one (**v01**). The shell is running **bash**. Its status is **s**, which means sleeping. Thiis is because it's waiting for the X Windows sytem to finish.



```
107 v01 S     0:00 sh /usr/X11/bin/startx
```

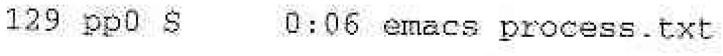
X Windows was started by the command **startx**. It won't finished until we exit from X. It too is sleeping.

```
115 v01 S     0:01 fvwm
```

The **fvwm** is a window manager for X, allowing other programs to be started and windows to be arranged on the screen.

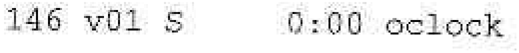
```
119 pp0 S     0:01 -bash
```

This process represents a window in the X Windows system. The shell, bash, is running in the new window. The window is running on a new pseudo terminal (/dev/ptyp0) abbreviated pp0.



```
129 pp0 S     0:06 emacs process.txt
```

This is the EMACS editor session started from the shell mentioned above. It uses the pseudo terminal.

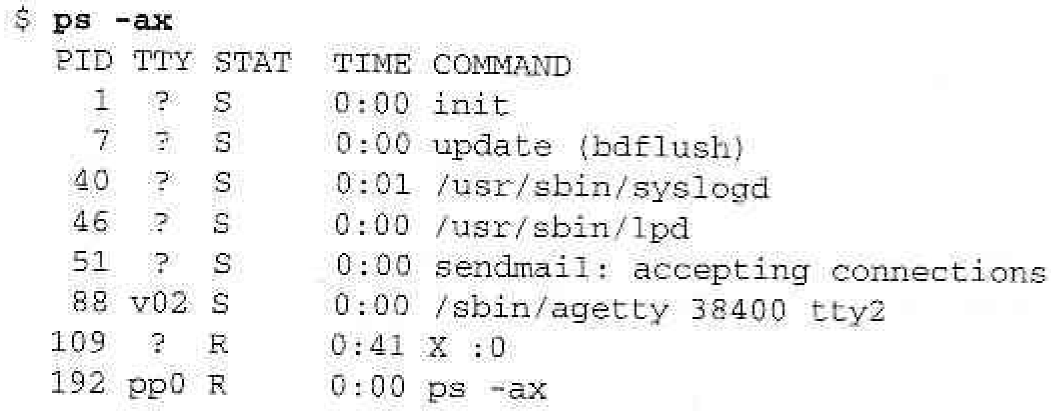


```
146 v01 S      0:00 oclock
```

This is a clock program started by the window manager. It's in the middle of a one-minute wait between updates of the clock hands.

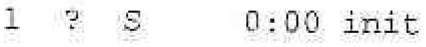
Process environment

Let's look at some other processes running on this Linux system. The output has been abbreviated for clarity:



```
$ ps -ax
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
   40  ?  S     0:01 /usr/sbin/syslogd
   46  ?  S     0:00 /usr/sbin/lpd
   51  ?  S     0:00 sendmail: accepting connections
   88 v02 S     0:00 /sbin/agetty 38400 tty2
  109  ?  R     0:41 X :0
  192 pp0 R     0:00 ps -ax
```

Here we can see one very important process indeed:



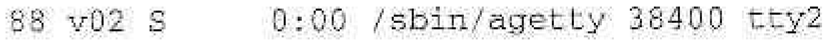
```
1  ?  S     0:00 init
```

In general, each process is started by another, known as its **parent process**. A process so started is known as a **child process**.

When UNIX starts, it runs a single program, the prime ancestror and process number one: **init**.

One such example is the login procedure **init** starts the **getty** program once for each terminal that we can use to long in.

These are shown in the **ps** output like this:



```
88 v02 S     0:00 /sbin/agetty 38400 tty2
```

When interacting with your server through a shell session, there are many pieces of information that your shell compiles to determine its behavior and access to resources. Some of these settings are contained within configuration settings and others are determined by user input.

One way that the shell keeps track of all of these settings and details is through an area it maintains called the environment. The environment is an area that the shell builds every time that it starts a session that contains variables that define system properties.

In this guide, we will discuss how to interact with the environment and read or set environmental and shell variables interactively and through configuration files. We will be using an Ubuntu 12.04 VPS as an example, but these details should be relevant on any Linux system.

Every time a shell session spawns, a process takes place to gather and compile information that should be available to the shell process and its child processes. It obtains the data for these settings from a variety of different files and settings on the system.

Basically the environment provides a medium through which the shell process can get or set settings and, in turn, pass these on to its child processes.

Environment List

The environment is implemented as strings that represent key-value pairs. If multiple values are passed, they are typically separated by colon (:) characters. Each pair will generally will look something like this:

```
KEY=value1:value2:...
```

If the value contains significant white-space, quotations are used:

```
KEY="value with spaces"
```

The keys in these scenarios are variables. They can be one of two types, environmental variables or shell variables.

Environmental variables are variables that are defined for the current shell and are inherited by any child shells or processes. Environmental variables are used to pass information into processes that are spawned from the shell.

Shell variables are variables that are contained exclusively within the shell in which they were set or defined. They are often used to keep track of ephemeral data, like the current working directory.

By convention, these types of variables are usually defined using all capital letters. This helps users distinguish environmental variables within other contexts.

Environment variables- getenv, setenv

Every process has an environment block that contains a set of environment variables and their values. There are two types of environment variables: user environment variables (set for each user) and system environment variables (set for everyone).

By default, a child process inherits the environment variables of its parent process. Programs started by the command processor inherit the command processor's environment variables. To specify a different environment for a child process, create a new environment block and pass a pointer to it as a parameter to the CreateProcess function.

The command processor provides the set command to display its environment block or to create new environment variables. You can also view or modify the environment variables by selecting System from the Control Panel, selectingAdvanced system settings, and clicking Environment Variables.

Each environment block contains the environment variables in the following format:

Var1=*Value1*\0
Var2=*Value2*\0
Var3=*Value3*\0
...
VarN=*ValueN*\0\0

The name of an environment variable cannot include an equal sign (=).

The GetEnvironmentStrings function returns a pointer to the environment block of the calling process. This should be treated as a read-only block; do not modify it directly. Instead, use the SetEnvironmentVariable function to change an environment variable. When you are finished with the environment block obtained from GetEnvironmentStrings,call the FreeEnvironmentStrings function to free the block. Calling SetEnvironmentVariable has no effect on the system environment variables.

Kernel support for process

The kernel runs the show, i.e. it manages all the operations in a Unix flavored environment. The kernel architecture must support the primary Unix requirements. These requirements fall in two categories namely, functions for process management and functions for file management (files include device files). Process management entails allocation of resources including CPU, memory, and offers services that processes may need. The file management in itself involves handling all the files required by processes, communication with device drives and regulating transmission of data to and from peripherals. The kernel operation gives the user processes a feel of synchronous operation, hiding all underlying asynchronism in peripheral and hardware operations (like the time slicing by clock). In summary, we can say that the kernel handles the following operations :

1. It is responsible for scheduling running of user and otherprocesses.

2. It is responsible for allocating memory.

3. It is responsible for managing the swapping between memory anddisk.

4. It is responsible for moving data to and from the peripherals.

5. it receives service requests from the processes and honors them.

Process Identification:

Every process has a unique process ID, a non-negative integer. There are two special processes. Process ID_0 is usually the schedule process and is often known as the ‘swapper’. No program on disk corresponds to this process – it is part of the kernel and is known as a system process, process ID_1 is usually the ‘init’ process and is invoked by the kernel at the end of the bootstrap procedure. The program files for this process loss /etc/init in older version of UNIX and is /sbin/init is newer version. ‘init’ usually reads the system dependent initialization files and brings the system to a certain state. The ‘init’ process never dies. ‘init’ becomes the parent process of any orphaned child process.

Process control

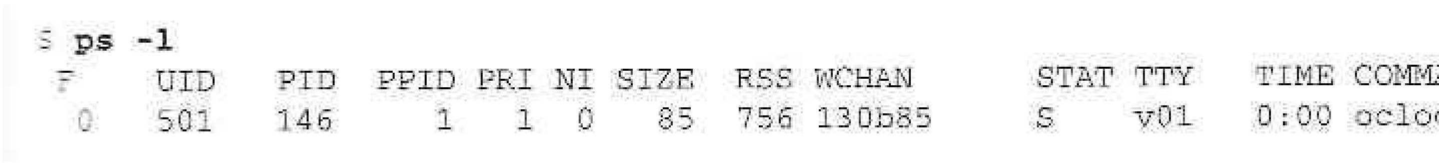
One further **ps** output example is the entry for the **ps** command itself:

```
192 pp0 R      0:00 ps -ax
```

This indicates that process 192 is in a run state (**R**) and is executing the command **ps-ax**.

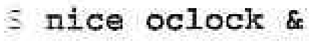
We can set the process priority using **nice** and adjust it using **renice**, which reduce the priority of a process by 10. High priority jobs have negative values.

Using the **ps -l** (forlong output), we can view the priority of processes. The value we are interested in is shown in the **NI** (nice) column:



```
$ ps -l
 F    UID   PID  PPID PRI NI SIZE  RSS WCHAN       STAT TTY   TIME COMM
  0   501   146     1   1  0   85  756 130b85      S    v01   0:00 oclo
```

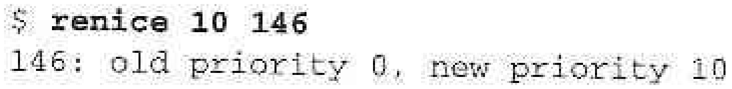
Here we can see that the **oclock** program is running with a default nice value. If it had been stated with the command,



```
$ nice oclock &
```

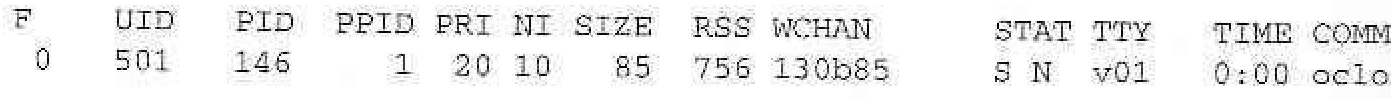
it would have been allocated a nice value of +10.

We can change the priority of a ruinning process by using the **renice** command,



```
$ renice 10 146
146: old priority 0, new priority 10
```

So that now the clock program will be scheduled to run less often. We can see the modified nice value with the **ps** again:



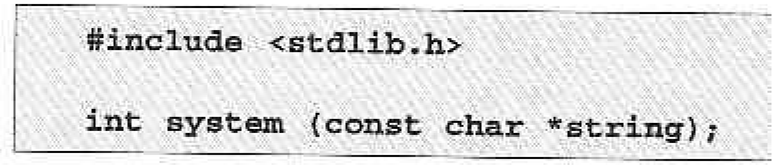
```
 F    UID   PID  PPID PRI NI SIZE  RSS WCHAN       STAT TTY   TIME COMM
  0   501   146     1  20 10   85  756 130b85      S N  v01   0:00 oclo
```

Notice that the status column now also contains **N**, to indicate that the nice value has changed from the default.

Process Creation

Starting New Processes

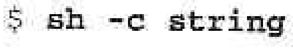
We can cause a program to run from inside another program and thereby create a new process by using the **system**. library function.



```
#include <stdlib.h>

int system (const char *string);
```

The **system** function runs the command passed to it as **string** and waits for it to complete.

The command is executed as if the command,

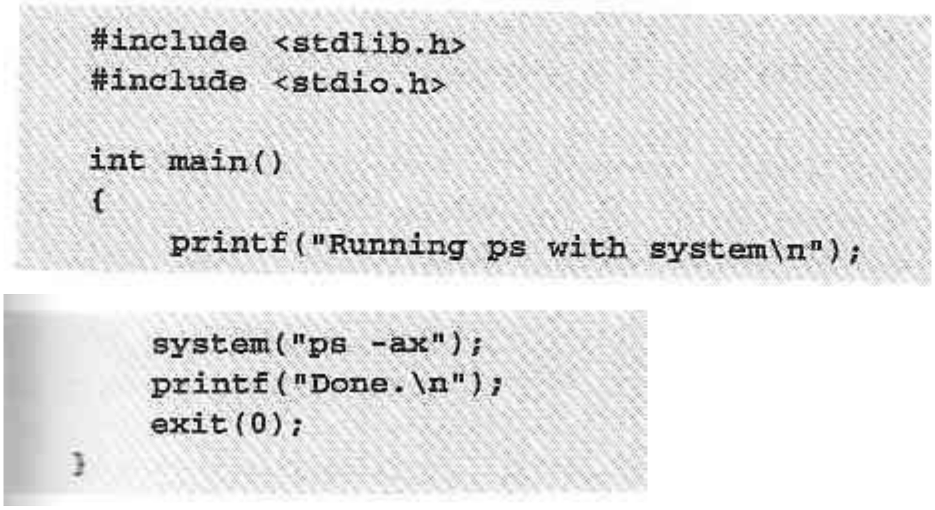


```
$ sh -c string
```

has been given to a shell.

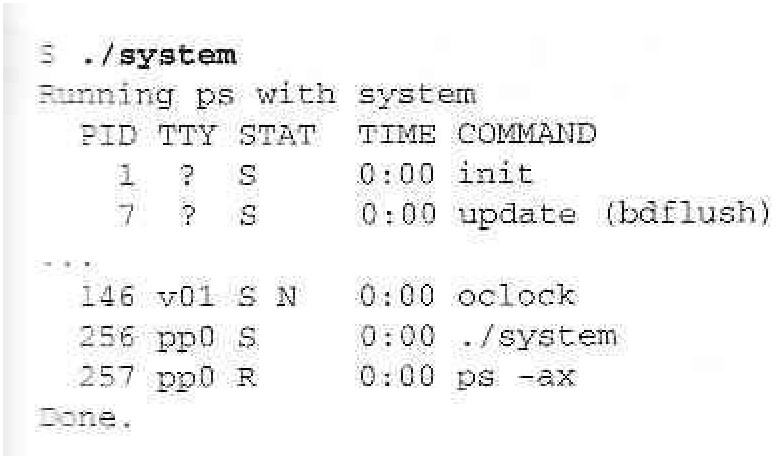
Try It Out - system

1. We can use **system** to write a program to run **ps** for us.



```
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("Running ps with system\n");
    system("ps -ax");
    printf("Done.\n");
    exit(0);
}
```

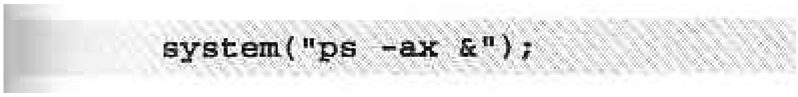
2. When we compile and run this program, **system.c**, we get thefollowing:



```
$ ./system
Running ps with system
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
...
  146 v01 S N   0:00 oclock
  256 pp0 S     0:00 ./system
  257 pp0 R     0:00 ps -ax
Done.
```

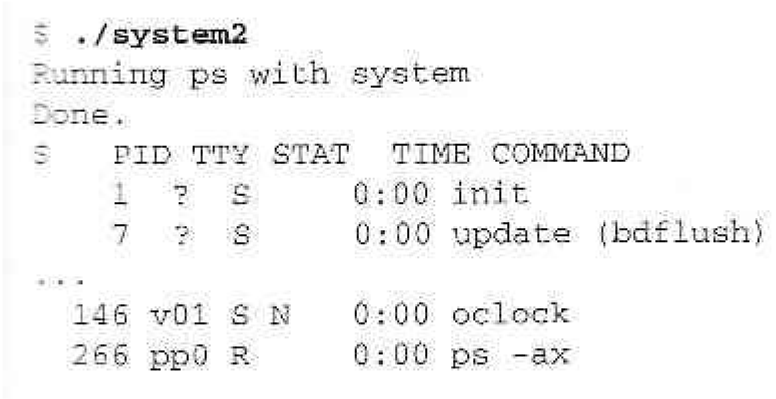
3. The **system** function uses a shell to start the desiredprogram.

We could put the task in the background, by changing the function call to the following:



```
        system("ps -ax &");
```

Now, when we compile and run this version of the program, we get:

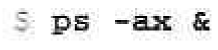


```
$ ./system2
Running ps with system
Done.
$   PID TTY STAT  TIME COMMAND
      1  ?  S     0:00 init
      7  ?  S     0:00 update (bdflush)
...
    146 v01 S N   0:00 oclock
    266 pp0 R     0:00 ps -ax
```

How It Works

In the first example, the program calls **system** with the string **"ps -ax"**, which executes the **ps** program. Our program returns from the call to **system** when the **ps** command is finished.

In the second example, the call to **system** returns as soon as the shell command finishes. The shell returns as soon as the **ps** program is started, just as would happen if we had typed,

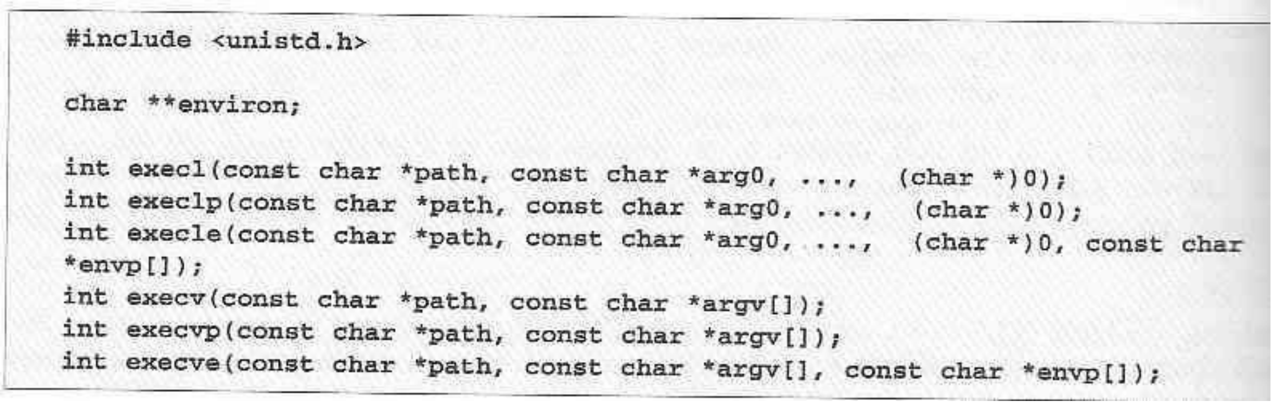


```
$ ps -ax &
```

at a shell prompt.

Replacing a Process Image

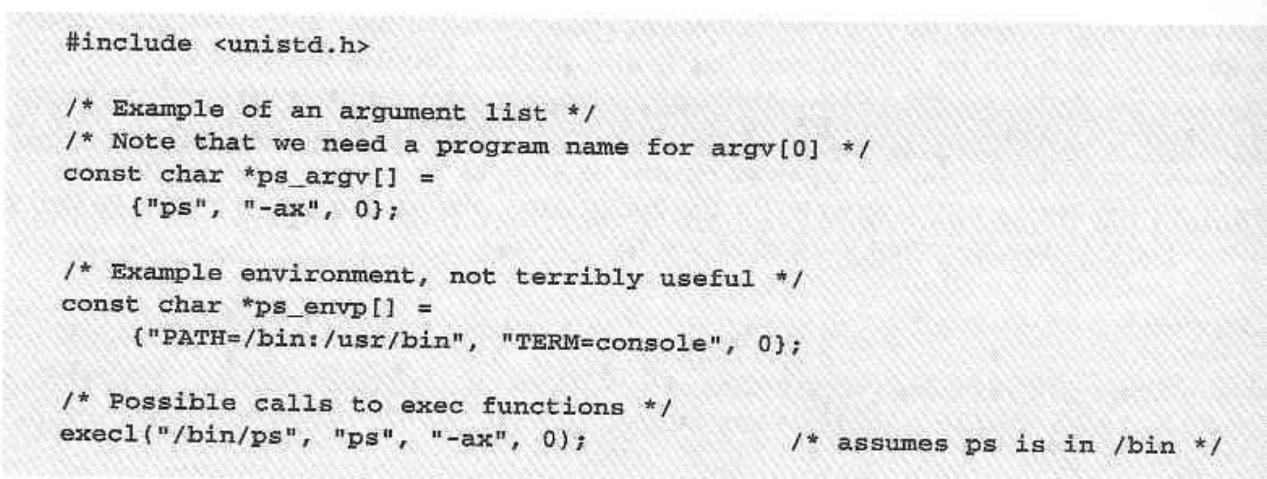
There is a whole family of related functions grouped under the **exec** heading. They differ in the way that they start processes and present program arguments.



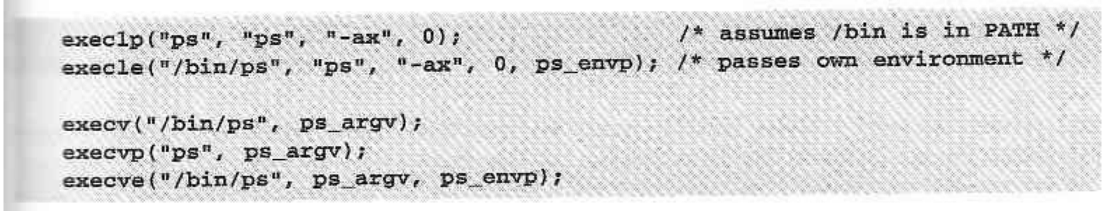
```
#include <unistd.h>

char **environ;

int execl(const char *path, const char *arg0, ...,  (char *)0);
int execlp(const char *path, const char *arg0, ...,  (char *)0);
int execle(const char *path, const char *arg0, ...,  (char *)0, const char
*envp[]);
int execv(const char *path, const char *argv[]);
int execvp(const char *path, const char *argv[]);
int execve(const char *path, const char *argv[], const char *envp[]);
```

The **exec** family of functions replace the current process with another created according to the arguments given.

If we wish to use an **exec** function to start the **ps** program as in our previous examples, we have the following choices:



```
#include <unistd.h>

/* Example of an argument list */
/* Note that we need a program name for argv[0] */
const char *ps_argv[] =
     {"ps", "-ax", 0};

/* Example environment, not terribly useful */
const char *ps_envp[] =
     {"PATH=/bin:/usr/bin", "TERM=console", 0};

/* Possible calls to exec functions */
execl("/bin/ps", "ps", "-ax", 0);              /* assumes ps is in /bin */
```



```
execlp("ps", "ps", "-ax", 0);                /* assumes /bin is in PATH */
execle("/bin/ps", "ps", "-ax", 0, ps_envp); /* passes own environment */

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

Try It Out - exclp

Let's modify our example to use an **exexlp** call.

```
#include <unistd.h>
#include <stdio.h>

int main()
{
    printf("Running ps with execlp\n");
    execlp("ps", "ps", "-ax", 0);
    printf("Done.\n");
    exit(0);
}
```

Now, when we run this program, **pexec.c**, we get the usual **ps** output, but no **Done**. message at all.

Note also that there is no reference to a process called **pexec** in the output:

```
$ ./pexec
Running ps with execlp
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
...
  146 v01 S N   0:00 oclock
  294 pp0 R     0:00 ps -ax
```

How It Works

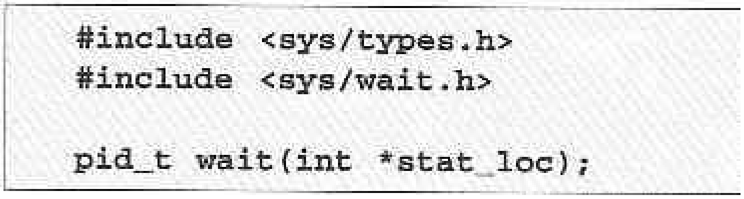
The program prints its first message and then calls **execlp**, which searches the directories given by the **PATH** environment variable for a program called **ps**.

It then executes this program in place of our **pexec** program, starting it as if we had given the shell command:

```
$ ps -ax
```

Waiting for a Process

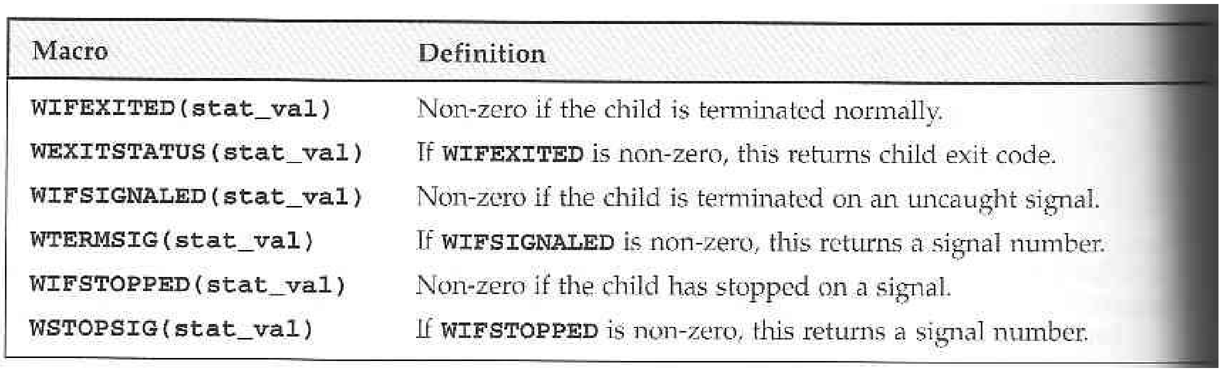
We can arrange for the parent process to wait until the child finishes before continuing by calling **wait**.



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

pid_t wait(int *stat_loc);
```

The **wait** system call causes a parent process to pause until one of its child processes dies or is stopped.

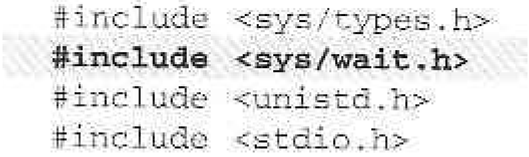
We can interrogate the status information using macros defined in **sys/wait.h**. These include:



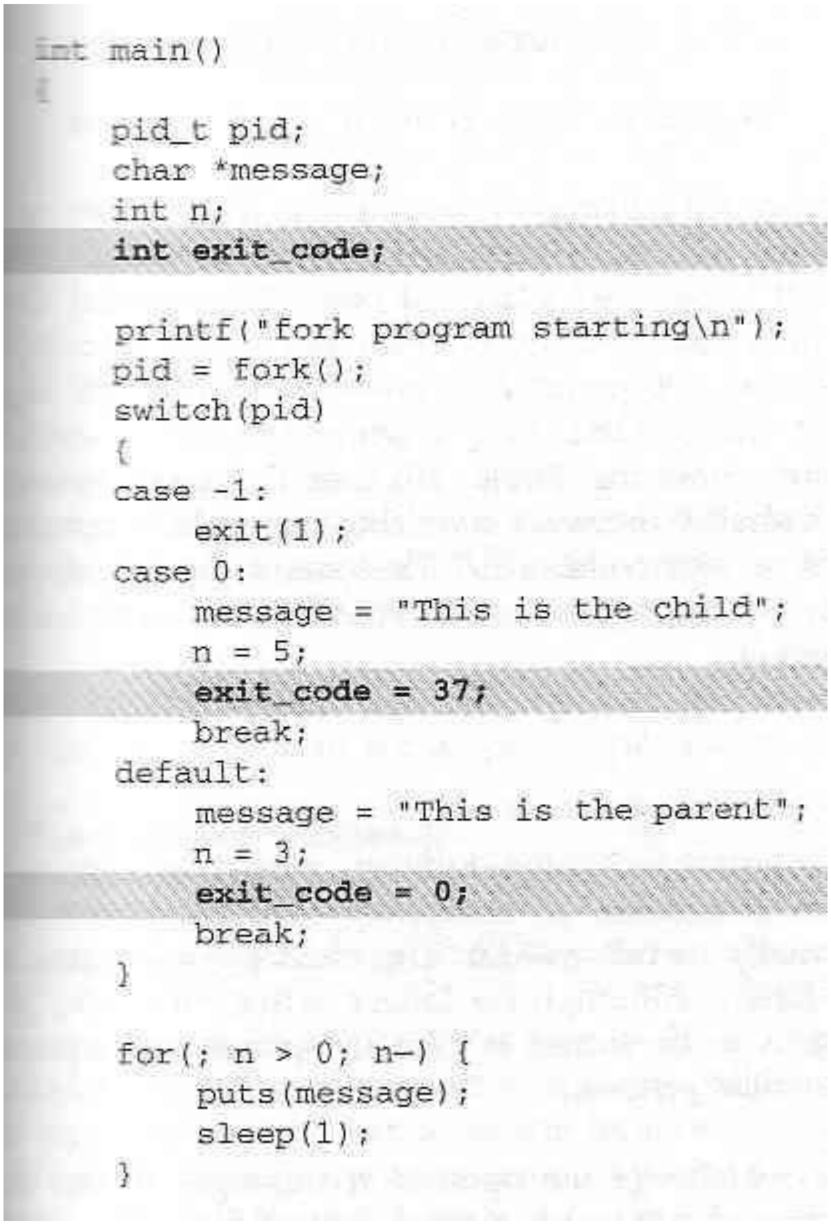
Macro	Definition
`WIFEXITED(stat_val)`	Non-zero if the child is terminated normally.
`WEXITSTATUS(stat_val)`	If `WIFEXITED` is non-zero, this returns child exit code.
`WIFSIGNALED(stat_val)`	Non-zero if the child is terminated on an uncaught signal.
`WTERMSIG(stat_val)`	If `WIFSIGNALED` is non-zero, this returns a signal number.
`WIFSTOPPED(stat_val)`	Non-zero if the child has stopped on a signal.
`WSTOPSIG(stat_val)`	If `WIFSTOPPED` is non-zero, this returns a signal number.

Try It Out - wait

1. Let's modify our program slightly so we can wait for and examine the child process exit status. Call the new program **wait.c**.

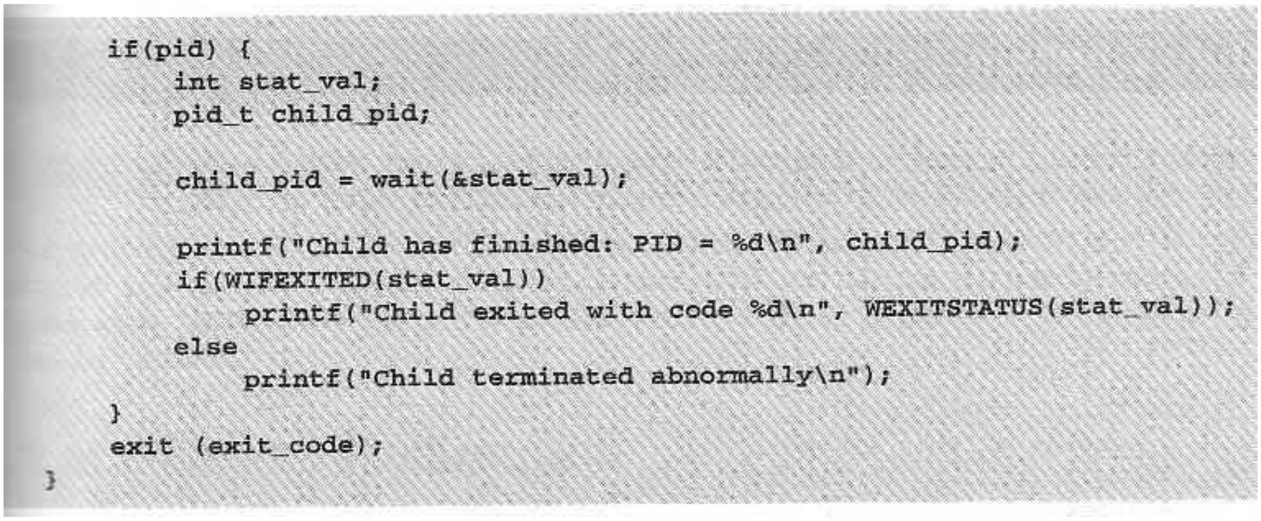


```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
```



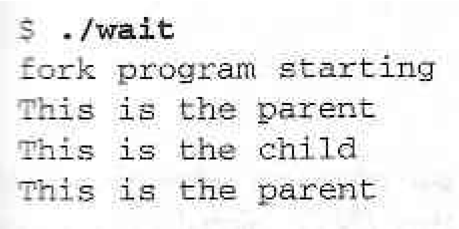
```
int main()
{
    pid_t pid;
    char *message;
    int n;
    int exit_code;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        exit_code = 37;
        break;
    default:
        message = "This is the parent";
        n = 3;
        exit_code = 0;
        break;
    }

    for(; n > 0; n–) {
        puts(message);
        sleep(1);
    }
```

2. This section of the program waits for the child process to finish:

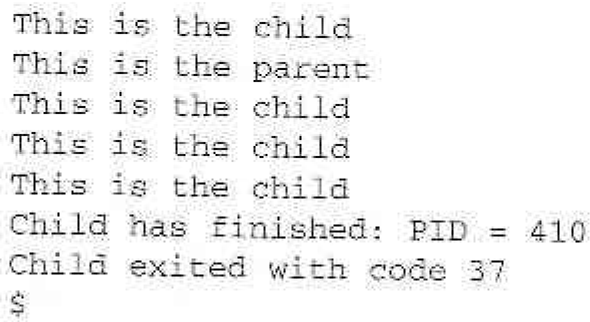


```
    if(pid) {
        int stat_val;
        pid_t child_pid;

        child_pid = wait(&stat_val);

        printf("Child has finished: PID = %d\n", child_pid);
        if(WIFEXITED(stat_val))
            printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else
            printf("Child terminated abnormally\n");
    }
    exit (exit_code);
}
```

When we run this program, we see the parent wait for the child. The output isn't confused and the exit code is reported as expected.



```
$ ./wait
fork program starting
This is the parent
This is the child
This is the parent
```



```
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 410
Child exited with code 37
$
```

How It Works

The parent process uses the **wait** system call to suspend its own execution until status information becomes available for a child process.

Zombie Processes

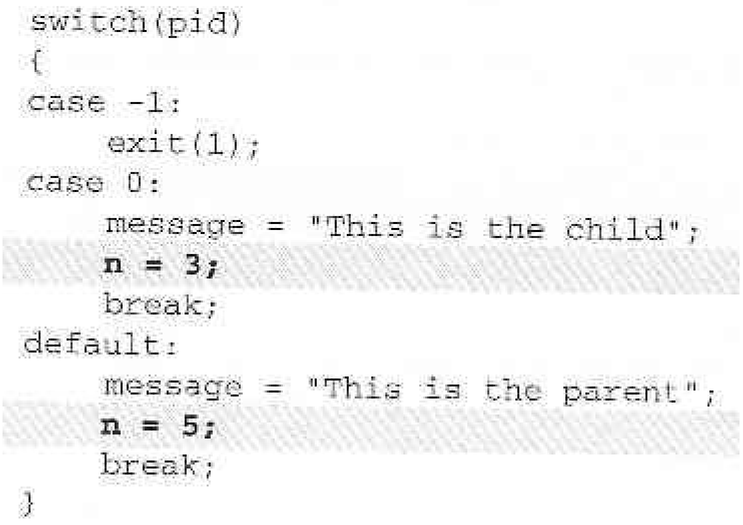
When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls **wait**.

This terminated child process is known as a **zombie process**.

Try It Out - Zombies

fork2.c is jsut the same as **fork.c**, except that the number of messages printed by the child and paent porcesses is reversed.

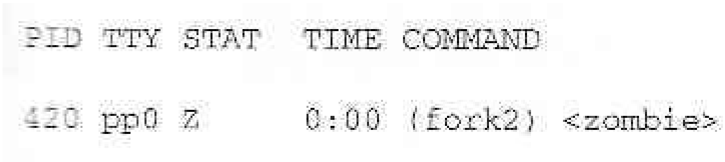
Here are the relevant lines of code:



```
switch(pid)
{
case -1:
     exit(1);
case 0:
     message = "This is the child";
     n = 3;
     break;
default:
     message = "This is the parent";
     n = 5;
     break;
}
```

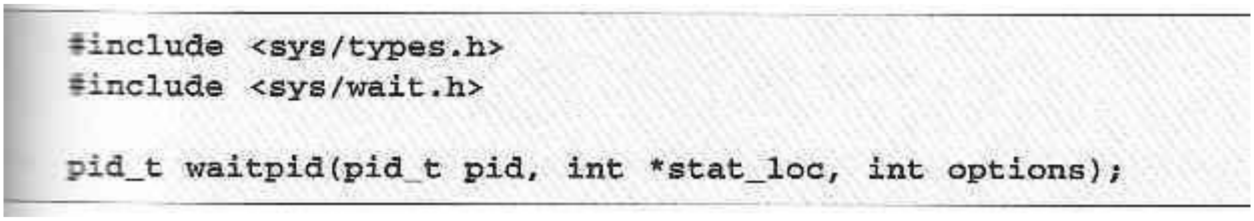
How It Works

If we run the above program with **fork2 &** and then call the **ps** program after the child has finished but before the parent has finished, we'll see a line like this:



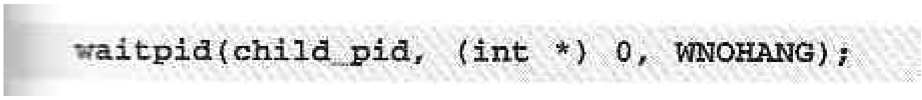
```
PID TTY STAT  TIME COMMAND

420 pp0 Z     0:00 (fork2) <zombie>
```

There's another system call that you can use to wail for child processes. It's called **waitpid** and youu can use it to wait for a specific process to terminate.



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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

If we want to have a parent process regularly check whether a specific child process had terminated, we could use the call,



```
waitpid(child_pid, (int *) 0, WNOHANG);
```

which will return zero if the child has not terminated or stopped or **child_pid** if it has.

Orphan Process

- When the parent dies first the child becomes **Orphan** .
- The kernel clears the process table slot for the parent.

System call interface for process management

In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers

#incldue<sys/types.h>

#include<unistd.h>

pid_t getpid(void);	Returns: process ID of calling process
pid_t geppid(void);	Returns: parent process ID OF calling process
uid_t getuid(void);	Returns: real user ID of calling process
uid_t geteuid(void);	Returns: effective user ID of calling process
gid_t getgid(void);	Returns: real group ID of calling process
gid_t getegid(void);	Returns: effective group ID of calling process

fork Function

The only way a new process is created by the UNIX kernel is when an existing process calls the fork function.

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

#include<unistd.h>

pid_t fork(void);
```

Return: 0 is child, process ID of child in parent, -1 on error

The new process created by fork is called child process. This is called once, but return twice that is the return value in the child is 0, while the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is because a process can have more than one child, so there is no function that allows a process to obtain the process IDs of its children. The reason fork return 0 to the child is because a process can have only a single parent, so that child can always call getppid to obtain the process ID of its parent.

Both the child and parent contain executing with the instruction that follows the call to fork. The child is copy of the parent. For example, the child gets a copy of the parent's data space, heap and stack. This is a copy for the child the parent and children don't share these portions of memory. Often the parent and child share the text segment, if it is read-only.

There are two users for fork:

1. When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. This is common for network servers_ the parent waits for a service requests from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
   When a process wants to execute a different program, this is common for shells. In this case the child does an exec right after it returns from the fork.

vfork Function

The function vfork has the same calling sequence and share return values as fork. But the semantics of the two functions differ. vfork is intended to create a new process when the purpose of the new process is to exec a new program. vfork creates the new process, just like fork, without fully copying the address space of the parent into the child, since the child won't reference the address space – the child just calls exec right after the vfork. Instead, while the child is running, until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual memory implementations of UNIX.

Another difference between the two functions is that vfork guarantees that the child runs first, until the parent resumes.

exit Function

There are three ways for a process to terminate normally, and two forms of abnormal termination.

1. Normal termination:
   a. Executing a return from the main function. This is equivalent to calling exit
   b. Calling the exit function
   c. Calling the _exit function
2. Abnormal termination
   a. Calling abort: It generates the SIGABRT signal
   b. When the process receives certain signals. The signal can be generated by the process itself

Regardless of how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and the like.

For any of the preceding cases we want the terminating process to be able to notify its parent how it terminated. For the exit and _exit functions this is done by passing an exit status as the argument to these two functions. In the case of an abnormal termination however, the kernel generates a termination status to indicate the reason for the abnormal termination. In any case, the parent of the process can obtain the termination status from either the wait or waitpid function.The exit status is converted into a termination status by the kernel when _exit is finally called. If the child terminated normally, then the parent can obtain the exit status of the child.

If the parent terminates before the child, then init process becomes the parent process of any process, whose parent terminates; that is the process has been inherited by init. Whenever a process terminates the kernel goes through all active processes to see if the terminating process is the parent of any process that still exists. If so, the parent process ID of the still existing process is changed to be 1 to assume that every process has a parent process.

When a child terminates before the parent, and if the child completely disappeared, the parent wouldn't be able to fetch its termination status, when the parent is ready to seek if the child had terminated. But parent get this information by calling wait and waitpid, which is maintained by the kernel.

wait and waitpid Functions

When a process terminates, either normally or abnormally, the parent is notified by the kernel sending the parent SIGCHLD signal. Since the termination of a child is an asynchronous event, this signal is the asynchronous notification from the kernel to the parent. The default action for this signal is to be ignored. A parent may want for one of its children to terminate and then accept it child's termination code by executing wait.

A process that calls wait and waitpid can

1. block (if all of its children are still running).
2. return immediately with termination status of a child (if a child has terminated and is waiting for its termination status to be fetched) or
3. return immediately with an error (if it down have any child process).

If the process is calling wait because it received SIGCHLD signal, we expect wait to return immediately. But, if we call it at any random point in time, it can block.

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

#include<sys/wait.h>
```

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);

Both return: process ID if OK, o or -1 on error

The difference between these two functions is

1. wait can block the caller until a child process terminates, while waitpid has an option that prevents it from blocking.
2. waitpid does not wait for the first child to terminate, it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates: if the caller blocks and has multiple children, wait returns when one terminates, we can know this process by PID return by the function.

For both functions, the argument statloc is pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

If we have more than one child, wait returns on termination of any of the children. A function that waits for a specific process is waitpid function.

The interpretation of the pid argument for waitpid depends on its value:

pid == -1 waits for any child process. Here, waitpid is equivalent to wait
pid > 0 waits for the child whose process ID equals pid
pid == 0 waits for any child whose process group ID equals that of the calling process
pid < -1 waits for any child whose process group ID equals the absolute value of pid

waitpid returns the process ID of the child that terminated, and its termination status is returned through statloc. With wait the only error is if the calling process has no children. With waitpid however, it's also possible to get an error if the specified process or process group does not exist or is not a child of the calling process.

The *options* argument lets us further control the operation of waitpid. This argument is either 0 or is constructed from the bitwise OR of the following constants.

WNOHANG	waitpid will not blink if a child specified by pid is not immediately available. In this case, the return value is 0.
WUNTRACED	if the status of any child specified by pid that has stopped, and whose status has not been updated since it has stopped, is returned

The waitpid function provides these features that are not provided by the wait function are:

1. waitpid lets us to wait for one particular process
2. waitpid provides a non-blocking version of wait
3. waitpid supports job control (wit the WUNTRACED option)

exec Function

The fork function can create a new process that then causes another program to be executed by calling one of the exec functions. When a process calls one of the exec functions, that process is completely replaced by the new program and the new program starts executing at its main function. The process ID doesn't change across an exec because a new process is not created. exec merely replaces the current process with a brand new program from disk.

There are six different exec functions. These six functions round out the UNIX control primitives. With fork we can create new processes, and with the exec functions we can initiate new programs. The exit function and the two wait functions handle termination and waiting for termination. These are the only process control primitives we need.

#include<unistd.h>

int execl(const char *pathname, const char *arg0, . . . /*(char *) 0*/

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, . . . /* (char *) 0, char envp[]*/);

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *pathname, const char *arg0, . . . /* (char *) 0*/);

int execvp(const char *filename, char *const argv[]);

All six returns: -1 on error, no return on success.

The first difference in these functions is that the first four take a pathname argument, while the last two take a filename argument. When a filename argument is specified:

- if filename contains a slash, it is taken as a pathname.
- Otherwise, the executable file is a searched for in directories specified by the PATH environment variable.

The PATH variable contains a list of directories (called path prefixes) that are separated by colors. For example, the name=value environment string

PATH=/bin:/usr/bin:usr/local/bin/:.

Specifies four directories to search, where last one is current working directory.

If either of the two functions, execlp or execvp finds an executable file using one of the path prefixes, but the file is not a machine executable that was generated by the link editor, it assumes the file is a shell script and tries to invoke /bin/sh with filename as input to the shell.

The next difference concerns the passing of argument list. The function execl, execlp and execle require each of the command-line arguments to the new program to be specified as separate arguments. The end of the argument should be a null pointer. For the other three functions execv, execvp and execve, we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

The final difference is the passing of the environment list to the new program. The two functions execle and execve allow us to pass a pointer to an array of pointer to an array of pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.

Differences Between Threads and Processes

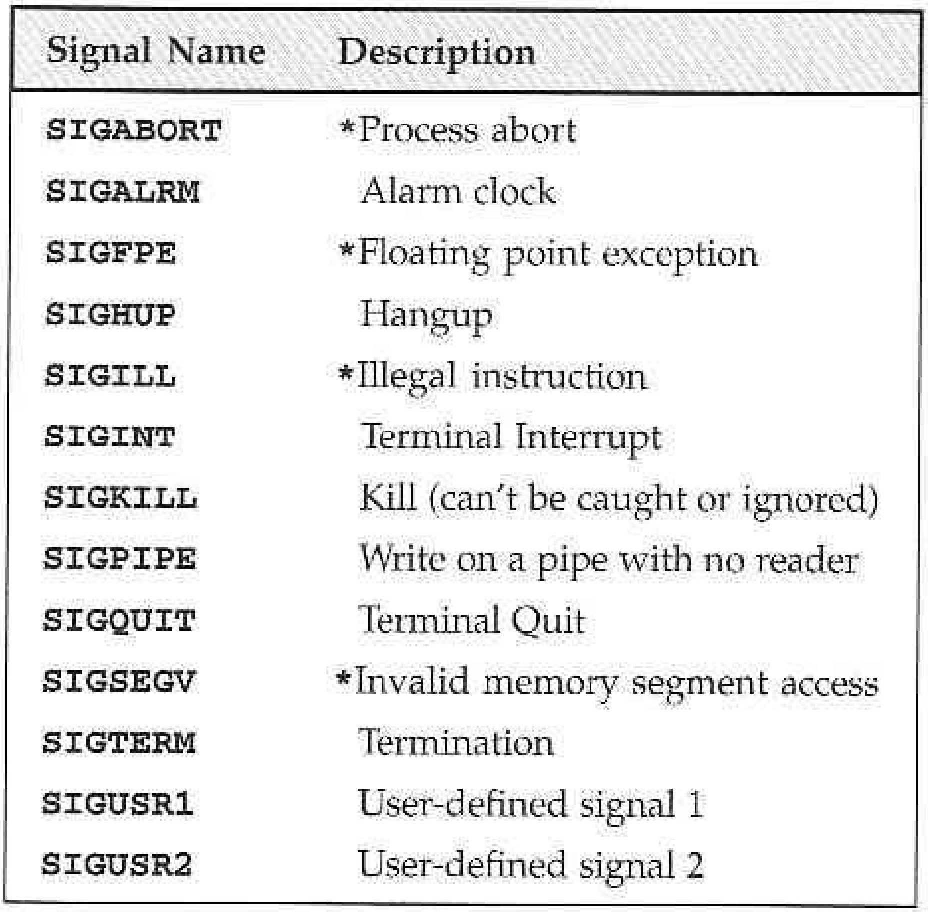
UNIX processes can cooperate; they can send each other messages and they can interrupt one another.

There is a class of process known as a **thread** which are distinct from processes in that they are separate execution streams within a single process.

Signals

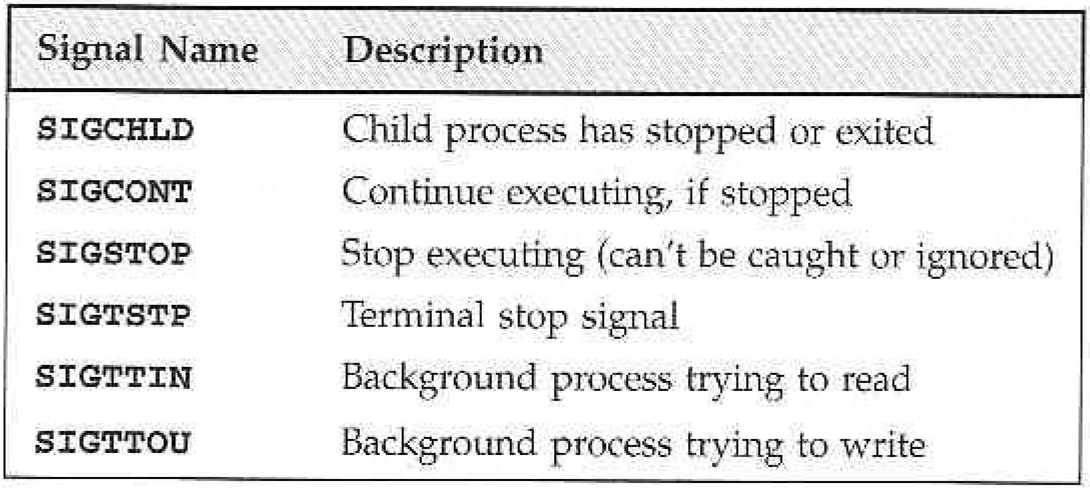
A **signal** is an event generated by the UNIX system in response to some condition, upon receipt of which a process may in turn take some action.

Signal names are defined in the header file **signal.h**. They all begin with **SIG** and include:



Signal Name	Description
SIGABORT	*Process abort
SIGALRM	Alarm clock
SIGFPE	*Floating point exception
SIGHUP	Hangup
SIGILL	*Illegal instruction
SIGINT	Terminal Interrupt
SIGKILL	Kill (can't be caught or ignored)
SIGPIPE	Write on a pipe with no reader
SIGQUIT	Terminal Quit
SIGSEGV	*Invalid memory segment access
SIGTERM	Termination
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2

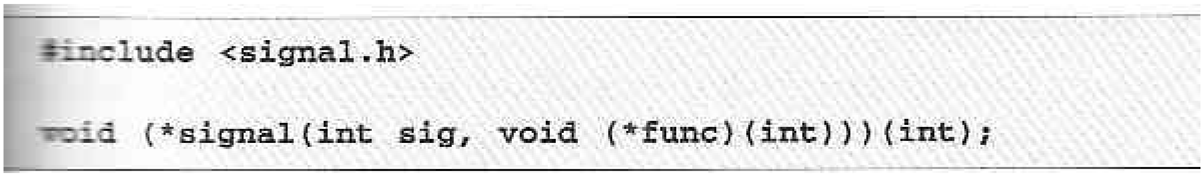
Additional signals include:



Signal Name	Description
SIGCHLD	Child process has stopped or exited
SIGCONT	Continue executing, if stopped
SIGSTOP	Stop executing (can't be caught or ignored)
SIGTSTP	Terminal stop signal
SIGTTIN	Background process trying to read
SIGTTOU	Background process trying to write

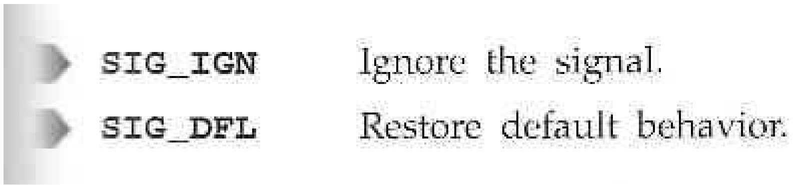
If the shell and terminal driver are configured normally, typing the interrupt character (Ctrl-C) at the keyboard will result in the **SIGINT** signal being sent to the foreground process. This will cause the program to terminate.

We can handle signals using the **signal** library function.



```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

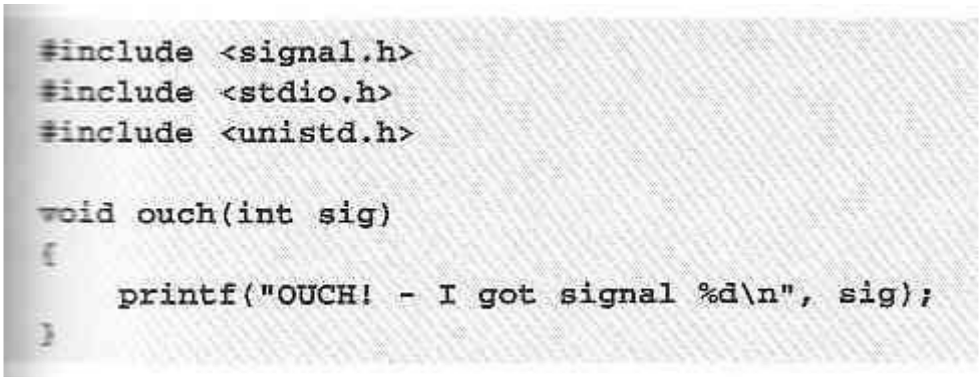
The **signal** function itself returns a function of the same type, which is the previous value of the function set up to handle this signal, or one of these tow special values:



SIG_IGN	Ignore the signal.
SIG_DFL	Restore default behavior.

Signal generation & Handling

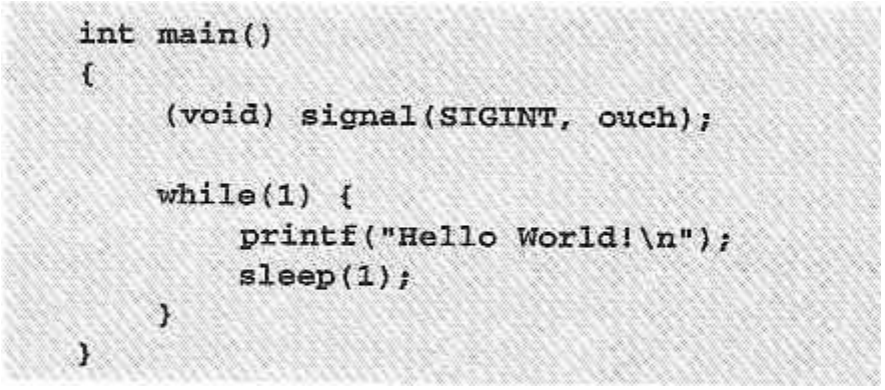
1. We'll start by writing the function which reacts to the signal which is passed in the parameter **sig**. Let's call it **ouch**:



```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}
```

2. The **main** function has to intercept the **SIGINT** signal generated when we typeCtrl-C.

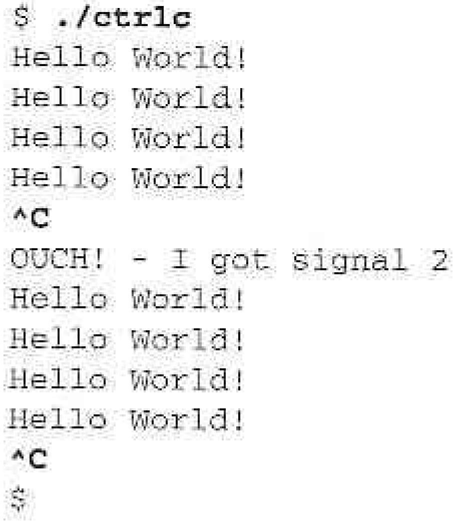
For the rest of the time, it just sits in an infinite loop, printing a message once a second:



```
int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

3. While the program is running, typing Ctrl-C causes it to react and then continue.

When we **type** Ctrl-C again, the program ends:



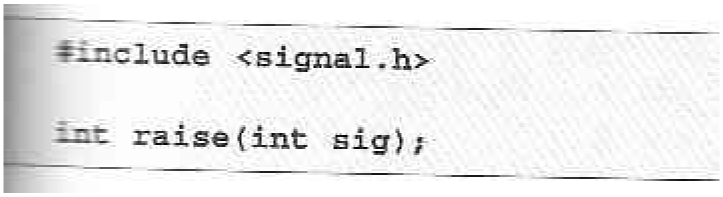
```
$ ./ctrlc
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
$
```

How It Works

The program arranges for the function **ouch** to be called when we type Ctrl-C, which gives the **SIGINT** signal.

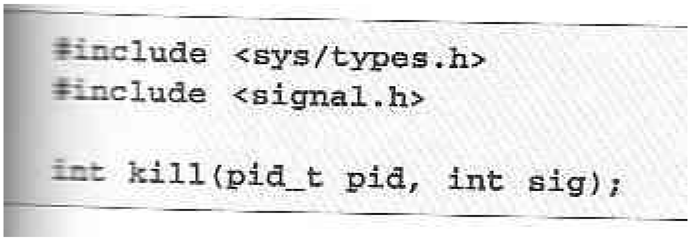
Kernel support for Signals-Sending Signals

A process may send a signal to itself by calling **raise**.



```
#include <signal.h>

int raise(int sig);
```

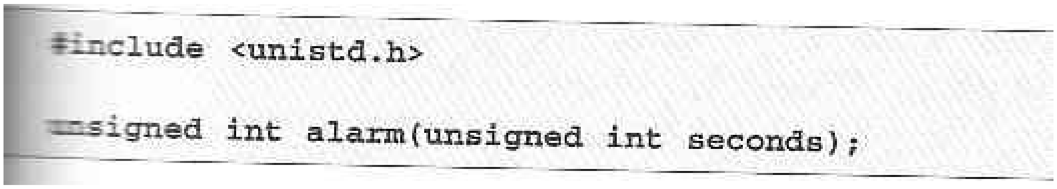
A process may send a signal to another process, including itself, by calling **kill**.



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

int kill(pid_t pid, int sig);
```

Signals provide us with a useful alarm clock facility.

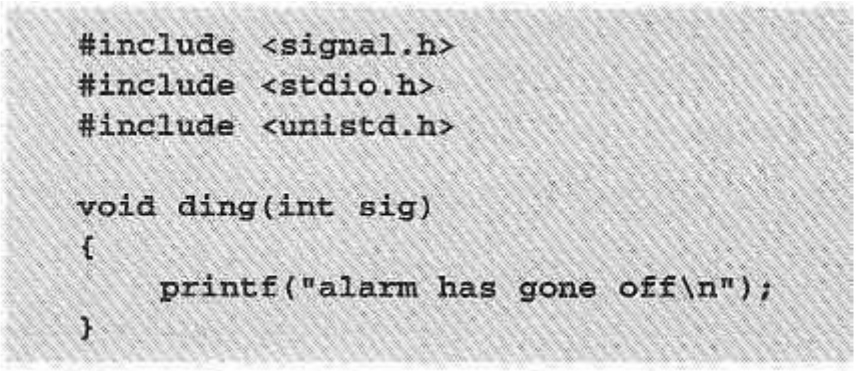
The **alarm** function call can be used by a process to schedule a **SIGALRM** signal at some time in the future.



```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

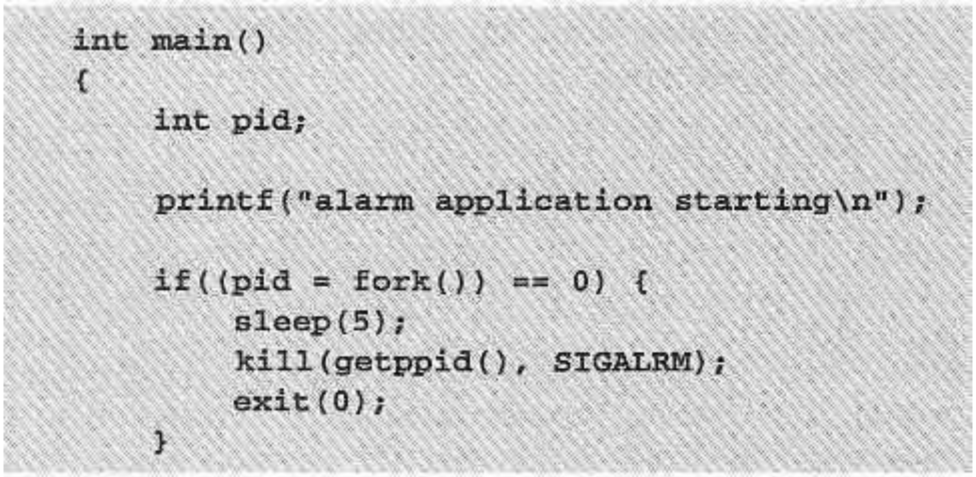
Try It Out - An Alarm Clock

1. In **alarm.c**, the first function, **ding**, simulates an alarm clock:



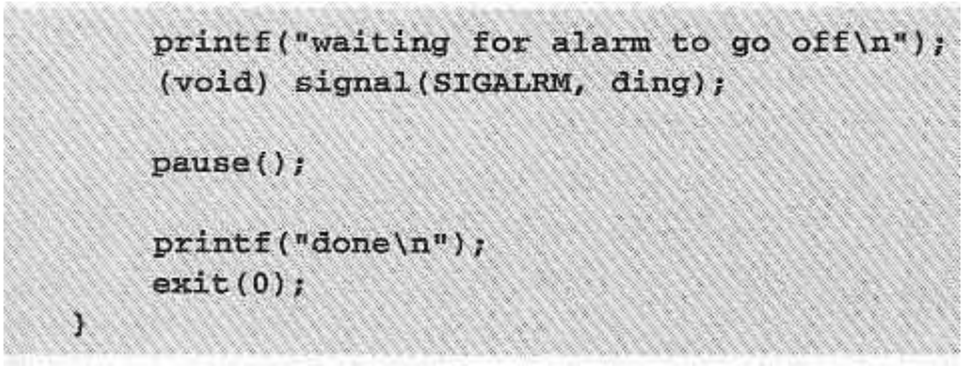
```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ding(int sig)
{
    printf("alarm has gone off\n");
}
```

2. In **main**, we tell the child process to wait for five seconds before sending a **SIGALRM** signal to its parent:



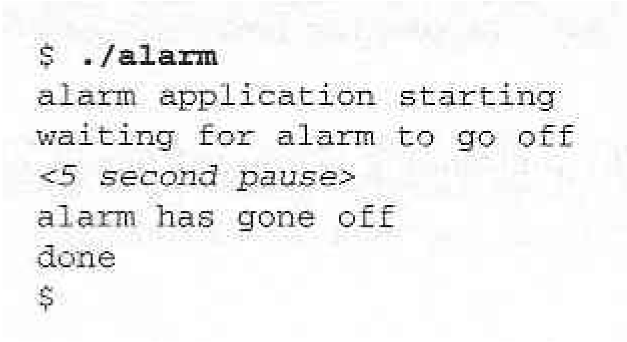
```
int main()
{
    int pid;

    printf("alarm application starting\n");

    if((pid = fork()) == 0) {
        sleep(5);
        kill(getppid(), SIGALRM);
        exit(0);
    }
```

3. The parent process arranges to catch **SIGALRM** with a call to **signal** and then waits forthe inevitable.



```
        printf("waiting for alarm to go off\n");
        (void) signal(SIGALRM, ding);

        pause();

        printf("done\n");
        exit(0);
    }
```

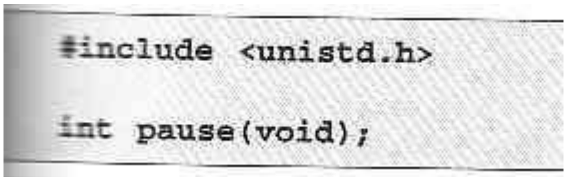
When we run this program, it pauses for five seconds while it waits for the simulated alarm clock.



```
$ ./alarm
alarm application starting
waiting for alarm to go off
<5 second pause>
alarm has gone off
done
$
```

This program introduces a new function, **pause**, which simply causes the program to suspend execution until a signal occurs.

It's declared as,



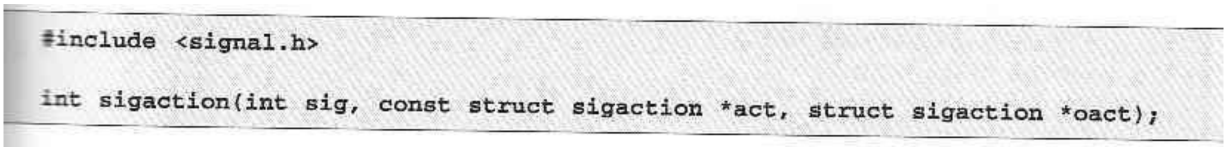
```
#include <unistd.h>

int pause(void);
```

How It Works

The alarm clock simulation program starts a new process via **fork**. This child process sleeps for five seconds and then sends a **SIGALRM** to its parent.

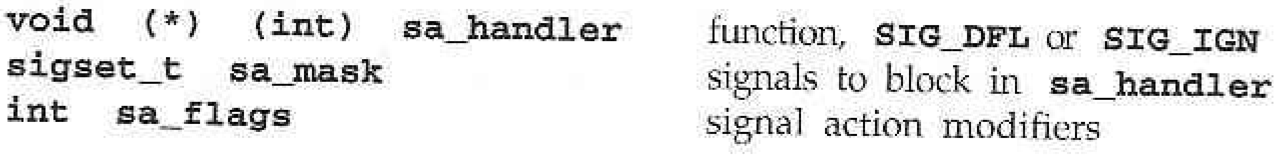
A Robust Signals Interface

X/Open specification recommends a newer programming interface for signals that is more robust: **sigaction**.



```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

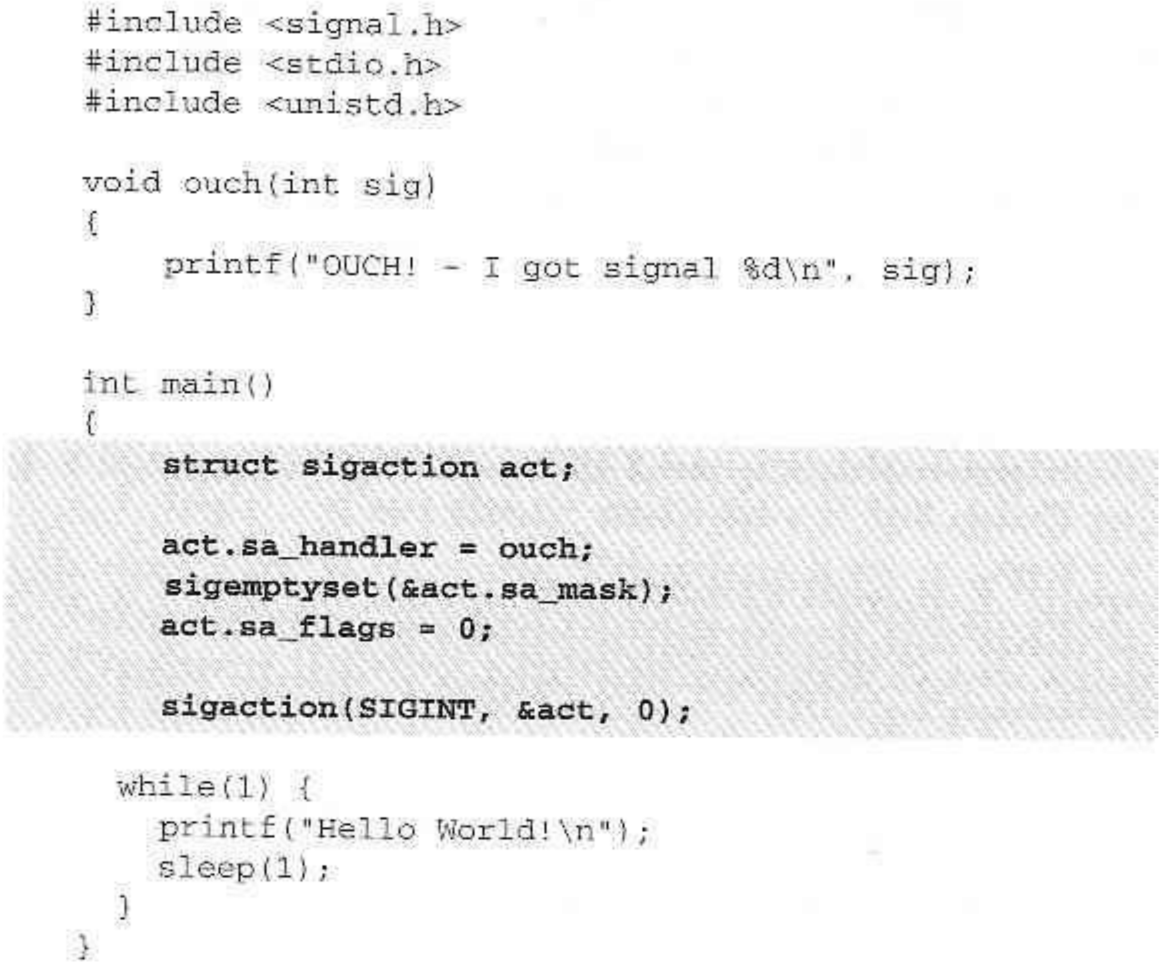
The **sigaction** structure, used to define the actions to be taken on receipt of the signal specified by **sig**, is defined in **signal.h** and has at least the following members:



`void (*) (int) sa_handler`	function, `SIG_DFL` or `SIG_IGN`
`sigset_t sa_mask`	signals to block in `sa_handler`
`int sa_flags`	signal action modifiers

Try It Out - sigaction

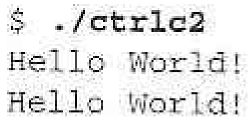
Make the changes shown below so that **SIGINT** is intercepted by **sigaction**. Call the new program **ctrlc2.c**.



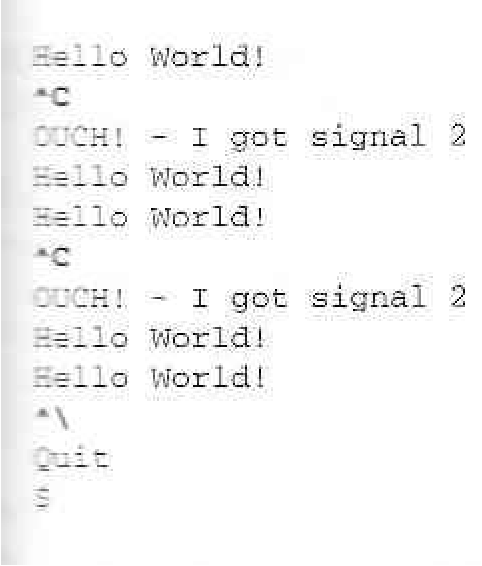
```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
    struct sigaction act;

    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, 0);

  while(1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

Running the program, we get a message when we type Ctrl-C because **SIGINT** is handled repeated;y by **sigaction**.

Type Ctrl-\ to terminate the program.



```
$ ./ctrlc2
Hello World!
Hello World!
```



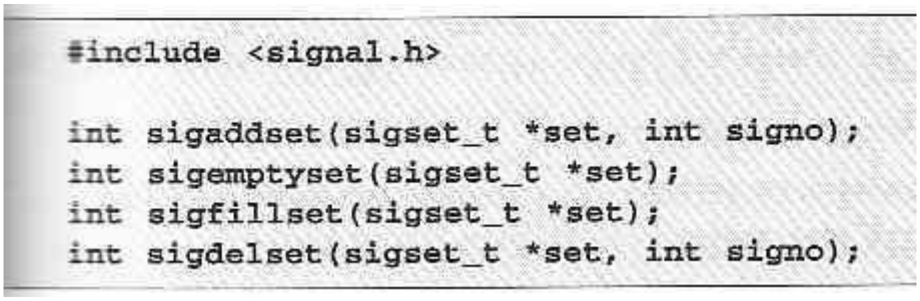
```
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^\
Quit
$
```

How It Works

The program calls **sigaction** instead of **signal** to set the signal handler for Ctrl-C (**SIGINT**) to the function **ouch**.

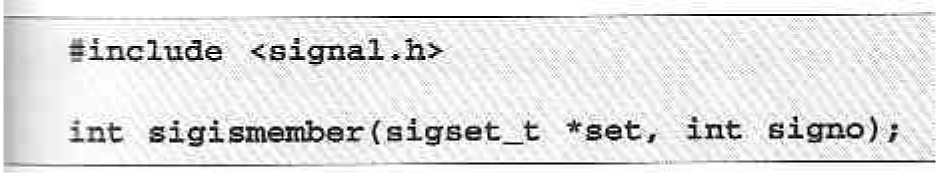
Signal Sets

The header file **signal.h** defines the type **sigset_t and functions used to manipulate sets of signals.**



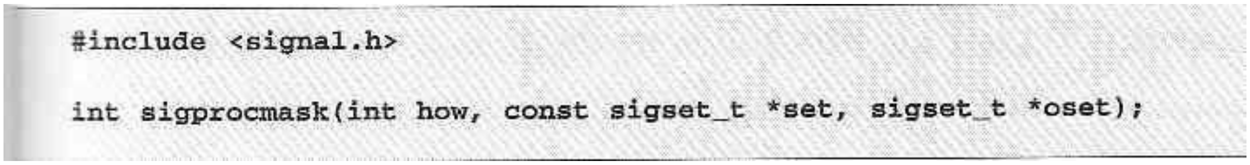
```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int signo);
```

The function **sigismember** determines whether the given signal is amember of a signal set.



```
#include <signal.h>

int sigismember(sigset_t *set, int signo);
```

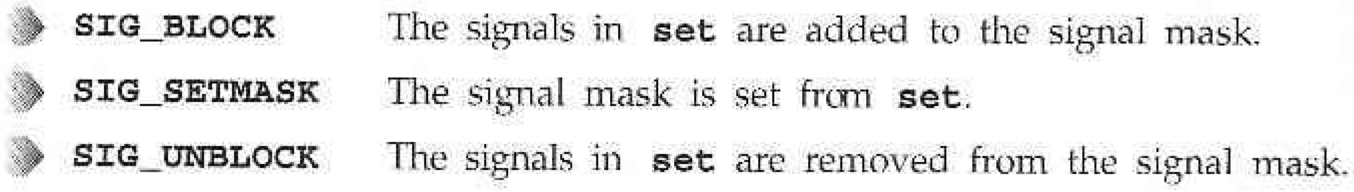
The process signal mask is set or examined by calling the function **sigprocmask**.



```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

sigprocmask can change the process signal mask in a number of ways according to the **how** argument.

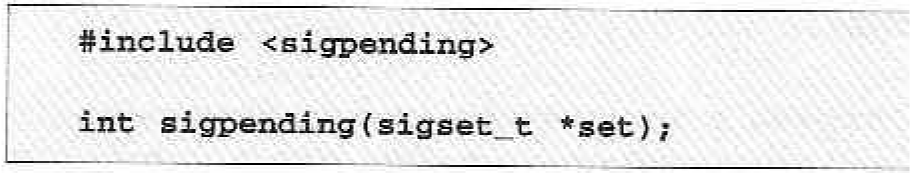
The **how** argument can be one of:



- **SIG_BLOCK** The signals in **set** are added to the signal mask.
- **SIG_SETMASK** The signal mask is set from **set**.
- **SIG_UNBLOCK** The signals in **set** are removed from the signal mask.

If a signal is blocked by a process, it won't be delivered, but will remain pending.

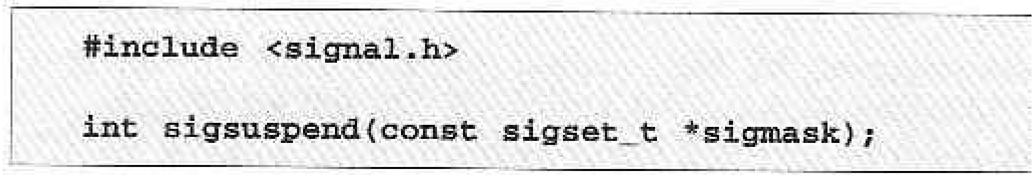
A program can determine which of its blocked signals ar pending by calling the function **sigpending**.



```
#include <sigpending>

int sigpending(sigset_t *set);
```

A process can suspend execution until the delivery of one of a set of signals by calling **sigsuspend**.

This is a more general form of the **pause** function we met earlier.



```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Signal Functions

The system calls related to signals are explained in the following sections.

Unreliable signals

The signals could get lost – a signal could occur and the process would never know about it. Here, the process has little control over a signal, it could catch the signal or ignore it, but blocking of a signal is not possible.

Reliable signals

Linux supports both POSIX reliable signals (hereinafter "standard signals") and POSIX real-time signals.

Signal dispositions

Each signal has a current *disposition*, which determines how the process behaves when it is delivered the signal.
The entries in the "Action" column of the tables below specify the default disposition for each signal.

kill and raise Functions

The kill function sends a signal to a process or a group of processes. The raise function allows a process to send a signal to itself.

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

#include<signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);
```

Both return: 0 if OK, -1 on error

There are four different conditions for the pid argument to kill:

pid > 0 The signal is sent to the process whose process ID is pid.

pid = 0 The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send thesignal.

pid<0 The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal.

pid = -1 unspecified.

alarm and pause Functions

The alarm function allows us to get a timer that will expire at a specified time in the future. When the timer expires, the SIGALRM signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

#include<unistd.h>

unsigned int alarm(unsigned int seconds);

Returns: 0 or number of seconds until previously set alarm.

The seconds value is the number of clock seconds in the future when the signal should be generated. There is only one of the alarm clocks per process. If, when we call alarm, there is a previously registered alarm clock for the process that has not yet expired, the number of seconds left for that alarm clock to return as the value of this function. That previously registered alarm clock is replaced by the new value.

If there is a previously registered alarm clock for the process that has not yet expired and if the *seconds* value is 0, the previous alarm clock is cancelled. The number of *seconds* left for that previous alarm clock is still returned as the value of the function.

Although the default action for SIGALRM is terminating the process, most processes use an alarm clock catch this signal.

The pause function suspends the calling process until a signal is caught.

```
#include<unistd.h>

int pause(void);
```

Returns: -1 with errno set to EINTR

The only time pause returns is if a signal handler is executed and that handler returns. In that case, pause returns -1 with errno set to EINTR.

abort Function

abort function causes abnormal program termination.

```
#include<stdlib.h>

void abort(void);
```

This function never returns.

This function sends the SIGABRT signal to the process. A process should not ignore this signal. abort overrides the blocking or ignoring of the signal by the process.

sleep Function

```
#include<unistd.h>

unsigned int sleep(unsigned int seconds);
```

Returns: 0 or number of unslept seconds.

This function causes the calling process to be suspended until either:

1. The amount of clock that is specified by *seconds* has elapsed or
2. A signal is caught by the process and the signal handlerreturns.

In case 1 the return value is 0 when sleep returns early, because of some signal being caught case 2, the return value is the number of unslept seconds.

Sleep can be implemented with an alarm function. If alarm is used, however, there can be interaction between the two functions.

Unix IPC

Unix has three major IPC constructs to facilitate interaction between processes:

- Message Queues (this PowerPoint document)
  - permit exchange of data between processes
- Semaphores
  - can be used to implement critical-section problems; allocation of resources
- Shared Memory
  - an area of memory accessible by multiple processes.

IPC System Calls

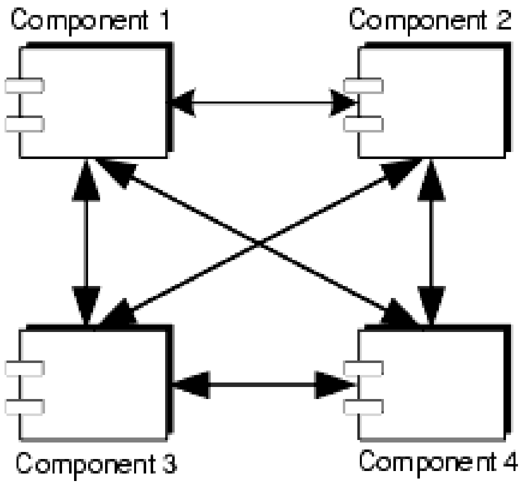
Functionality	Message Queue	Semaphore	Shared
Allocate IPC	***msgget***	***semget***	***shmget***
Access IPC	***msgsnd*** ***msgrcv***	***semop***	***shmat*** ***shmdt***
IPC Control	***msgctl***	***semctl***	***shmctl***

Message Queues

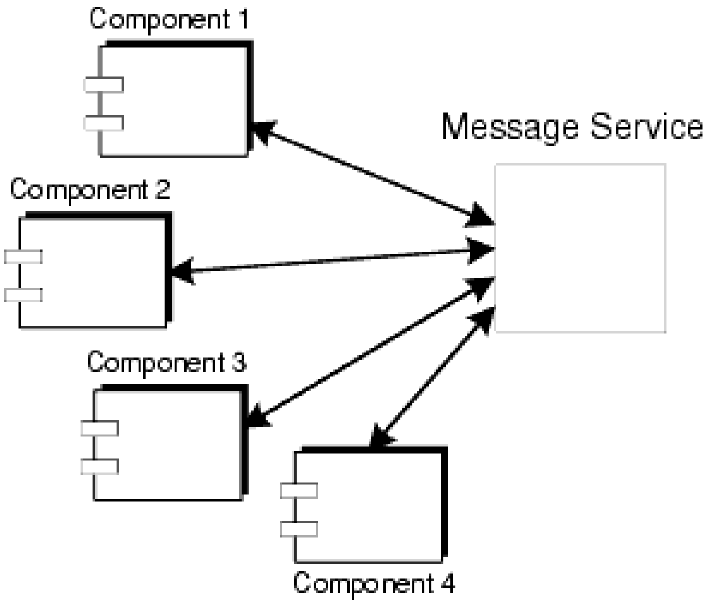
- Creating a Message Queue
- Message Queue Control
- Message Queue Operations
- IPC Call
- Client-Server Example

Messaging Methods

Peer to Peer Messaging



Centralized Messaging



Message Queues

- One process establishes a message queue that others may access. Often a server will establish a message queue that multiple clients can access
- Features of Message Queues
  - A process generating a message may specify its type when it places the message in a message queue.
  - Another process accessing the message queue can use the message type to selectively read only messages of specific type(s) in a first-in-first-out manner.
  - Message queues provide a user with a means of multiplexing data from one or more producer(s) to one or more consumer(s).

Attributes of Message Queues

- A conceptual view of a message queue, from *Interprocess Communications in Unix*:
  - The attributes of each element on the queue:

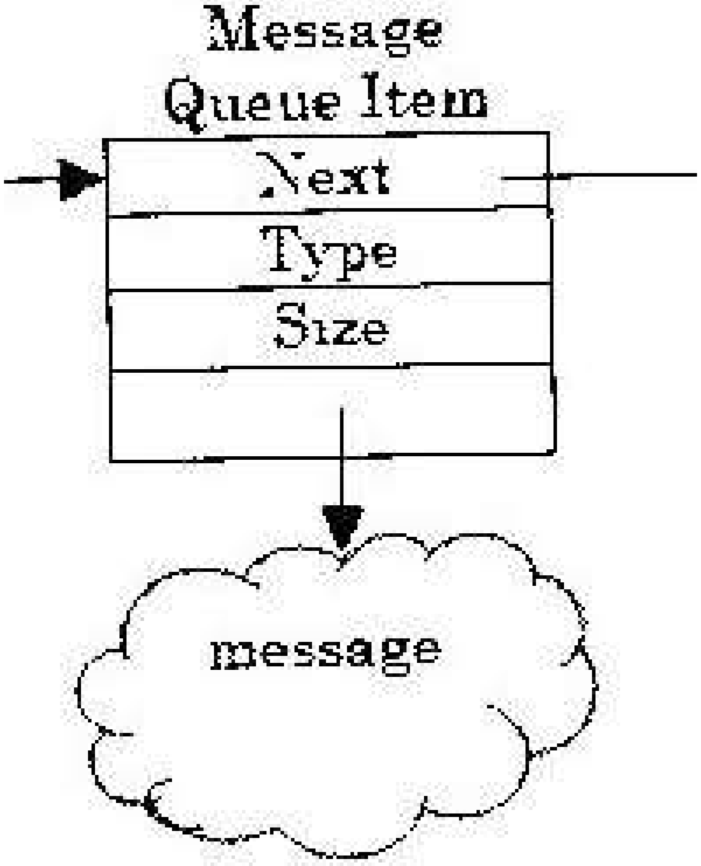
- long integer ***type***;

- ***size*** of the data portion of the message (can be zero);

* A message queue element then has one more field:

- ***data*** (if the length is greater than zero)

* Message Structure



Message Queue Structure

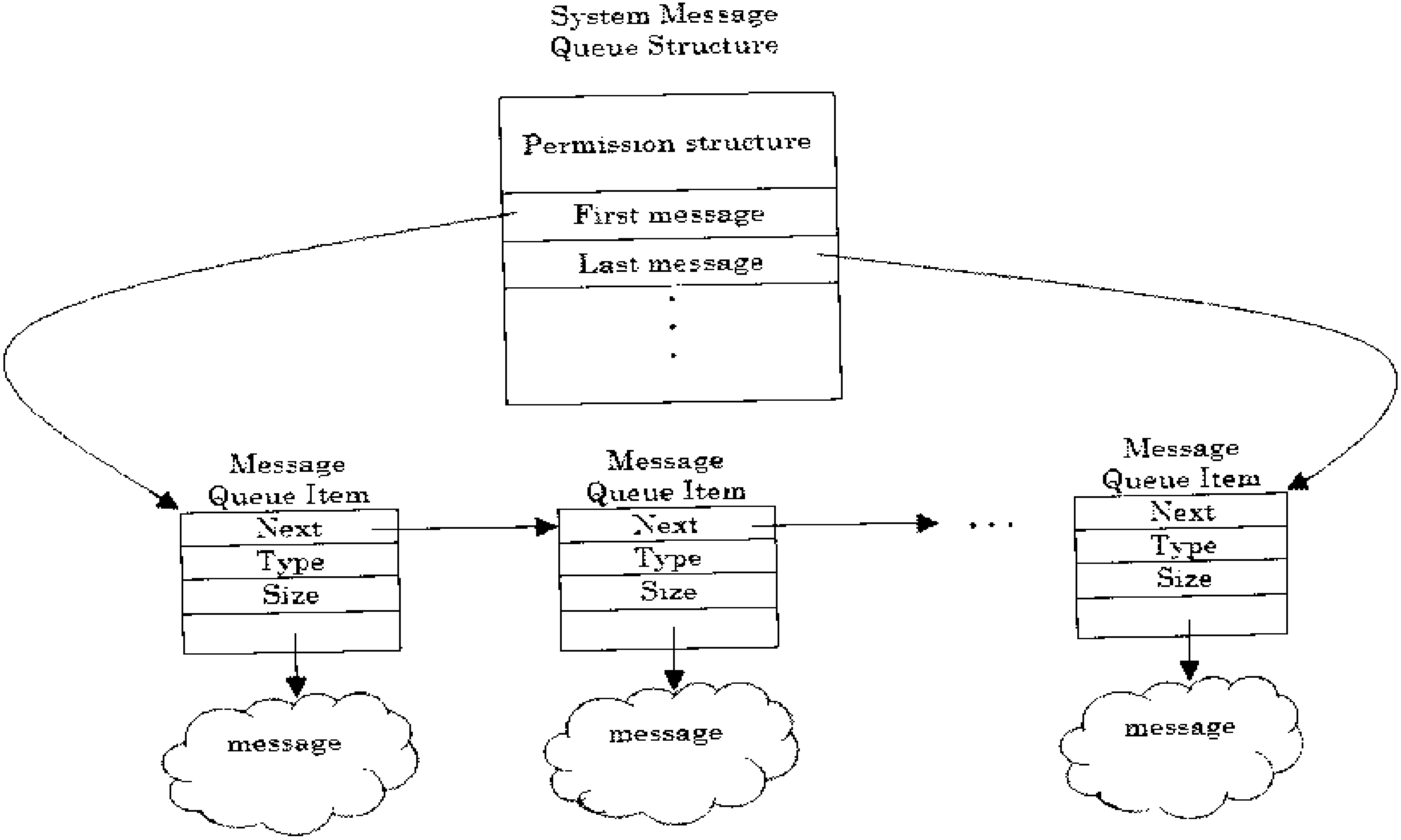


Figure 6.5 A message queue with *N* items.

msqid_ds Structure

```
struct  msqid_ds {
    struct ipc_perm  msg_perm; /*operation permission struct */
    struct  msg  *msg_first;  /* pointer to first message on q*/
    struct  msg  *msg_last;  /* point to last message on q */
    ulong            msg_cbytes; /* current # bytes on q  */
    ulong            msg_qnum;  /* # of message on q */
    ulong            msg_qbytes; /* max # of bytes on q */
    pid_t            msg_lspid;     /* pid of last msgsnd */
    pid_t            msg_lrpid;    /* pid of last msgrcv */
    time_t         msg_stime;    /* last msgsnd time  */
   ...........................
};   /* total 17 members */
```

ipc_perm Structure

```
*   struct  ipc_perm {
        uid_t           uid;      /*owner’s user ID */
        gid_t           gid;      /* owner’s group ID */
        uid_t           cuid;     /* creator’s user ID */
        gid_t           cgid;     /* creator’s group ID */
        mode_t          mode;     /* access modes */
        ulong           seg;      /* slot usage sequence number */
        key_t           key;      /* key */
        long            pad[4];          /* reserve area */
    };
* Struct  msqid_ds {
    struct          ipc_perm        msg_perm;..... };
```

msg structure

```
struct   msg  {
    struct  msg    *msg_next;  /* pointer to next message on q */
    long              msg_type;  /*  message type */
    ushort            msg_ts;     /*  message text size */
    short             msg_spot;  /* message text map address */
};
```

msgget System Call

Create a message queue.

- Includes: <sys/types.h> <sys/ipc.h> <sys/msg.h>
- Command: ***int msgget(key_t key, int msgflg);***
  - Returns: Success: message queue identifier ;

    Failure: -1;
  - Arguments:
    - ***key***: to be specified directly by the user or generated using ftok. We will use a function getuid() to generate unique, consistent message queues for each person
    - ***msgflg***: IPC_CREAT, IPC_EXCL or permission value.

Example of generating a message queue: Qcreate.cpp

msgsnd System Call (Message Queue Operation)

- Function: to place (send) message in the message queue.
- Include: <sys/types.h> <sys/ipc.h> <sys/msg.h>
- Summary:

  int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg)

  - Returns: Success: 0; Failure: -1; Sets errno: Yes
  - Arguments
    - ***int msgid***: valid message queue identifier
    - ***const void *msgp***: address of the message to be sent
    - ***size_t msgsz***: the size of the message to be sent.
    - ***int msgflg***: Two possible values:
      - 0: Block, if the message queue is full
      - IPC_NOWAIT : don't wait if message queue is full

msgrcv System Call (Message Queue Operation)

- Function: to retrieve message from the message queue.
  - Include <sys/types.h> <sys/ipc.h> <sys/msg.h>
  - ***int msgrcv (int msqid, void *msgp,***
    size_t msgsz, long msgtyp, int msgflg);
    - Return: Success: number of bytes actually received;
      Failure: -1; Sets errno: Yes
- Arguments:
  - ***int msqid***: the message queue identifier.
  - ***void *msgp***: a point to received message location (structure).
  - ***size_t msgsz***: the maximum size of the message in bytes.
  - ***long msgtype***: the type of the message to be retrieved.
  - ***int msgflg***: to indicate what action should be taken.
    - 0: error if size of message exceeds msgsz
    - MSG_NOERROR: if size of message exceeds msgsz, accept msgsz bytes
    - IPC_NOWAIT: return –1 with errno set to ENOMSG

Type of Message (*msgrcv* System Call)

int msgrcv (int msqid, void *msgp, size_t   msgsz,
long msgtyp, int   msgflg);

- ***long msgtype***: the type of the message to be retrieved.
  - Actions for ***msgrcv*** as indicated by ***msgtyp*** value

msgtyp value	action
0	return first (oldest) message on queue
> 0	return first message with type equal to msgtyp
< 0	return the first message with lowest type less than or equal to msgtyp

Remove a Message Queue in a Program

Command:

msgctl (msqid, IPC_RMID, (struct msqid_ds *) 0);

- To remove the message queue with key *msqid*.
- You must be the owner

Fork and Exec Unix Model

Tutorial 3

Process Management Model

The Unix process management model is split into two distinct operations :

1. The creation of a process.
2. The running of a new program.

Process Management Model

- The creation of a new process is done using the **fork()** system call.
- A new program is run using the **exec**(l,lp,le,v,vp) family of system calls.
- These are two separate functions which may be used independently.

The Fork() System Call

1. A call to fork() will create a completely separate sub-process which will be exactly the same as the parent.
2. The process that initiates the call to fork is called the parent process.
3. The new process created by fork is called the child process.
4. The child gets a copy of the parent's text and memory space.

5. They do not share the same memory.

Fork return values

fork() system call returns an integer to both the parent and child processes:

- -1 this indicates an error with no child process created.
- A value of zero indicates that the child process code is being executed.
- Positive integers represent the child’s process identifier (PID) and the code being executed is in the parent’s process.

Simple Fork Example

```
            if ( (pid = fork()) == 0)
            printf( “I am the child\n ”);
else
            printf( “I am the parent\n ”);
```

The exec() System Call

Calling one of the **exec()** family will terminate the currently running program and starts executing a new one which is specified in the parameters of exec in the context of the existing process. The process id is not changed.

Exec family of functions

- int **execl**(const char *path, const char *arg, ...);
- int **execlp**(const char *file, const char *arg, ...);
- int **execle**(const char *path, const char *arg , ..., char * const envp[]);
- int **execv**(const char *path, char *const argv[]);
- int **execvp**(const char *file, char *const argv[]);

Exec family of functions

The first difference in these functions is that the first four take a pathname argument while the last two take a filename argument. When a filename argument is specified:

- if filename contains a slash, it is taken as a pathname.
- otherwise the executable file is searched for in the directories specified by the PATH environment variable.

Simple Execlp Example

```
#include <sys/type.h>
#include <stdio.h>
#include <unistd.h> int
main()
{
        pid_t pid;
        /* fork a child process */
        pid = fork();
        if (pid < 0){ /* error occurred */
                fprintf(stderr, “Fork Failed”);
                exit(-1);
        }
        else if (pid == 0){ /* child process */
                execlp(“/bin/ls”,”ls”,NULL);
        }
        else { /* parent process */
                /* parent will wait for child to complete */
                wait(NULL); printf(“Child
                Complete”); exit(0);
        }
}
```

When a program wants to have another program running in parallel, it will typically first use **fork**, then the child process will use **exec** to actually run the desired program.

The **fork-and-exec** mechanism switches an old command with a new, while the environment in which the new program is executed remains the same, including configuration of input and output devices, and environment variables.

Semaphores

When we write programs that use threads, operating in either multiuser systems, multiprocessing systems or a combination of the two, we often discover that we have **critical sections** of code where we need to ensure that a single executing process has exclusive access to a resource.

In our first example application using dbmto access a database in Chapter 7, the data could be corrupted if multiple programs tried to update the database at exactly the same time. There's no trouble with two different programs asking different users to enter data for the database, the only problem is in the parts of the code that update the database. These are the critical sections.

To prevent problems caused by more than one program accessing a resource, we need a way of generating and using a token that grants access to only one thread of execution in a critical section at a time. We saw briefly in Chapter 11 some thread specific ways we could use a mutex or semaphores to control access to critical sections in a threaded program. In this chapter we return to the topic of semaphores, but look more generally at how they are used between different processes. Be aware that the semaphore functions used with threads are not the ones we will meet here, be careful not to confuse the two types.

It's surprisingly difficult to write code that achieves this without specialist hardware support, although there's a solution known as **Dekker's Algorithm**. Unfortunately, this algorithm relies on a 'busy wait', or 'spin lock', where a process runs continuously, waiting for a memory location to be changed. In a multitasking environment such as UNIX, this is an undesirable waste of CPU resources.

One possible solution that we've already seen is to create files using the O_EXCL flag with the open function, which provides atomic file creation. This allows a single process to succeed in obtaining a token, the newly created file. This method is fine for simple problems, but rather messy and very inefficient for more complex examples.

An important step forward in this area of concurrent programming occurred when Dijkstra introduced the concept of the **semaphore**. A semaphore is a special variable that takes only whole positive numbers and upon which only two operations are allowed: wait and signal. Since 'wait' and 'signal' already have special meanings in UNIX programming, we'll use the original notation:

- P(semaphore variable) for wait,
- V(semaphore variable) for signal.

These letters come from the Dutch words for wait (*passeren*: to pass, as in a check point before the critical section) and signal (*vrijgeven*: to give, as in giving up control of the critical section). You may also come across the terms 'up' and 'down' used in relation to semaphores, taken from the use of signaling flags.

Semaphore Definition

Semaphore Definition

The simplest semaphore is a variable that can take only the values 0 and 1, a **binary semaphore**, and this is the most common form. Semaphores that can take many positive values are called **general semaphores**. For the remainder of this chapter, we'll concentrate on binary semaphores.

The definitions of P and V are surprisingly simple. Suppose we have a semaphore variable, sv. The two operations are then defined as:

P(sv)	If sv is greater than zero, decrement sv. If sv is zero, suspend execution of this process.
V(sv)	If some other process has been suspended waiting for sv, make it resume

	execution. If no process is suspended waiting for sv, increment sv.

Another way of thinking about semaphores is that the semaphore variable, sv, is true when the critical section is available, is decremented by P(sv) so it's false when the critical section is busy, and incremented by V(sv) when the critical section is again available. Be aware that simply having a normal variable that you decrement and increment is not good enough, because you can't express in C or C++ the need to make a single, atomic operation of the test to see whether the variable is true, or the change of the variable to make it false. This is what makes the semaphore operations special.

A Theoretical Example

We can see how this works with a simple theoretical example. Suppose we have two processes proc1 and proc2, both of which need exclusive access to a database at some point in their execution. We define a single binary semaphore, sv, that starts with the value 1 and can be accessed by both processes. Both processes then need to perform the same processing to access the critical section of code, indeed the two processes could simply be different invocations of the same program.

The two processes share the sv semaphore variable. Once one process has executed P(sv), it has obtained the semaphore and can enter the critical section. The second process is prevented from entering the critical section because, when it attempts to execute P(sv), it's made to wait until the first process has left the critical section and executed V(sv)to release the semaphore.

The required pseudo−code is:

```
semaphore sv = 1;
loop forever {
   P(sv);
   critical code section;
   V(sv);
   non−critical code section;
}
```

The code is surprisingly simple because the definition of the P and V operations is very powerful. Here's a diagram showing how the P and V operations act as a gate into critical sections of code:

UNIX System V API’s for Semaphores(semget,semctl,semop)

NAME

(i) semget - get a semaphore set identifier

SYNOPSIS

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

#include <sys/ipc.h>

#include <sys/sem.h>



intsemget(key_t key, int nsems, int semflg);
```

DESCRIPTION

The **semget**() system call returns the semaphore set identifier associated with the argument *key*. A new set of *nsems*semaphores is created if *key* has the value **IPC_PRIVATE** or if no existing semaphore set is associated with *key* and**IPC_CREAT** is specified in *semflg*.

If *semflg* specifies both **IPC_CREAT** and **IPC_EXCL** and a semaphore set already exists for *key*, then **semget**() fails with *errno* set to **EEXIST**. (This is analogous to the effect of the combination **O_CREAT | O_EXCL** for**open**(2).)

Upon creation, the least significant 9 bits of the argument *semflg* define the permissions (for owner, group and others) for the semaphore set. These bits have the same format, and the same meaning, as the *mode* argument of**open**(2) (though the execute permissions are not meaningful for semaphores, and write permissions mean permission to alter semaphore values).

The values of the semaphores in a newly created set are indeterminate. (POSIX.1-2001 is explicit on this point.) Although Linux, like many other implementations, initialises the semaphore values to 0, a portable application cannot rely on this: it should explicitly initialise the semaphores to the desired values.

When creating a new semaphore set, **semget**() initialises the set's associated data structure, *semid_ds* (see**semctl**(2)), as follows:

Tag	Description
	sem_perm.cuid and *sem_perm.uid* are set to the effective user ID of the calling process.
	sem_perm.cgid and *sem_perm.gid* are set to the effective group ID of the calling process.
	The least significant 9 bits of *sem_perm.mode* are set to the least significant 9 bits of *semflg*.
	sem_nsems is set to the value of *nsems*.
	sem_otime is set to 0.
	sem_ctime is set to the current time.

The argument *nsems* can be 0 (a don't care) when a semaphore set is not being created. Otherwise *nsems* must be greater than 0 and less than or equal to the maximum number of semaphores per semaphore set (**SEMMSL**).

If the semaphore set already exists, the permissions are verified.

RETURN VALUE

If successful, the return value will be the semaphore set identifier (a nonnegative integer), otherwise -1 is returned, with *errno* indicating the error.

ERRORS

On failure *errno* will be set to one of the following:

Tag	Description

EACCES	A semaphore set exists for *key*, but the calling process does not have permission to access the set, and does not have the **CAP_IPC_OWNER** capability.
EEXIST	A semaphore set exists for **key** and *semflg* specified both **IPC_CREAT** and**IPC_EXCL**.
EINVAL	*nsems* is less than 0 or greater than the limit on the number of semaphores per semaphore set (**SEMMSL**), or a semaphore set corresponding to *key* already exists, and *nsems* is larger than the number of semaphores in that set.
ENOENT	No semaphore set exists for *key* and *semflg* did not specify **IPC_CREAT**.
ENOMEM	A semaphore set has to be created but the system does not have enough memory for the new data structure.
ENOSPC	A semaphore set has to be created but the system limit for the maximum number of semaphore sets (**SEMMNI**), or the system wide maximum number of semaphores (**SEMMNS**), would be exceeded.

NAME

(ii) semctl - semaphore control operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

intsemctl(int_semid_**, int**_semnum_**, int**_cmd_**, ...);**

DESCRIPTION

semctl() performs the control operation specified by *cmd* on the semaphore set identified by *semid*, or on the*semnum*-th semaphore of that set. (The semaphores in a set are numbered starting at 0.)

This function has three or four arguments, depending on *cmd*. When there are four, the fourth has the type *union semun*. The *calling program* must define this union as follows:

```
union semun {
intval;    /* Value for SETVAL */
structsemid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
   unsigned short  *array;  /* Array for GETALL, SETALL */
structseminfo  *__buf;  /* Buffer for IPC_INFO
                         (Linux specific) */
};
```

The *semid_ds* data structure is defined in <sys/sem.h> as follows:

```
structsemid_ds {

structipc_permsem_perm;  /* Ownership and permissions

time_tsem_otime; /* Last semop time */
time_tsem_ctime; /* Last change time */

    unsigned short  sem_nsems; /* No. of semaphores in set */

};
```

The *ipc_perm* structure is defined in <sys/ipc.h> as follows (the highlighted fields are settable using **IPC_SET**):

```
structipc_perm {
key_t key;            /* Key supplied to semget() */
uid_tuid;            /* Effective UID of owner */
gid_tgid;            /* Effective GID of owner */
uid_tcuid;           /* Effective UID of creator */
gid_tcgid;           /* Effective GID of creator */
    unsigned short mode;  /* Permissions */
    unsigned short seq;   /* Sequence number */
};
```

Valid values for *cmd* are:

Tag	Description
IPC_STAT	Copy information from the kernel data structure associated with *semid* into the*semid_ds* structure pointed to

	by *arg.buf*. The argument *semnum* is ignored. The calling process must have read permission on the semaphore set.
IPC_SET	Write the values of some members of the *semid_ds* structure pointed to by *arg.buf* to the kernel data structure associated with this semaphore set, updating also its*sem_ctime* member. The following members of the structure are updated:*sem_perm.uid*, *sem_perm.gid*, and (the least significant 9 bits of) *sem_perm.mode*. The effective UID of the calling process must match the owner (*sem_perm.uid*) or creator (*sem_perm.cuid*) of the semaphore set, or the caller must be privileged. The argument *semnum* is ignored.
IPC_RMID	Immediately remove the semaphore set, awakening all processes blocked in **semop**() calls on the set (with an error return and *errno* set to **EIDRM**). The effective user ID of the calling process must match the creator or owner of the semaphore set, or the caller must be privileged. The argument *semnum* is ignored.
IPC_INFO (Linux specific)	
	Returns information about system-wide semaphore limits and parameters in the structure pointed to by *arg.__buf*. This structure is of type *seminfo*, defined in<*sys/sem.h*> if the _GNU_SOURCE feature test macro is defined: structseminfo { intsemmap; /* # of entries in semaphore map; unused */ intsemmni; /* Max. # of semaphore sets */ intsemmns; /* Max. # of semaphores in all

```
                   semaphore sets */
   intsemmnu;  /* System-wide max. # of undo
                   structures; unused */
   intsemmsl;  /* Max. # of semaphores in a set */
   intsemopm;  /* Max. # of operations for semop() */
   intsemume;  /* Max. # of undo entries per
                   process; unused */
   intsemusz;  /* size of structsem_undo */
   intsemvmx;  /* Maximum semaphore value */
   intsemaem;  /* Max. value that can be recorded for
                   semaphore adjustment (SEM_UNDO) */
   };
```

The *semmsl*, *semmns*, *semopm*, and *semmni* settings can be changed via*/proc/sys/kernel/sem*; see **proc**(5) for details.

SEM_INFO (Linux specific)	
	Returns a *seminfo* structure containing the same information as for **IPC_INFO**, except that the following fields are returned with information about system resources consumed by semaphores: the *semusz* field returns the number of semaphore sets that currently exist on the system; and the *semaem* field returns the total number of semaphores in all semaphore sets on the system.
SEM_STAT (Linux specific)	
	Returns a *semid_ds* structure as for **IPC_STAT**. However, the *semid* argument is not a semaphore identifier, but instead an index into the kernel's internal array that maintains information about all semaphore sets on the system.
GETALL	Return **semval** (i.e., the current value) for all semaphores of the set into *arg.array***.**The argument *semnum* is ignored. The calling process must have read permission on the semaphore set.
GETNCNT	The system call returns the value of **semncnt** (i.e., the number of processes waiting for the value of this semaphore to increase) for the *semnum*-th semaphore of the set (i.e. the number of processes waiting for an increase of **semval** for the *semnum*-th semaphore of the set). The calling process must have read permission on the semaphore set.
GETPID	The system call returns the value of **sempid** for the *semnum*-th semaphore of the set (i.e. the PID of the process that executed the last **semop**() call for the *semnum*-th semaphore of the set). The calling process must have read permission on the semaphore set.
GETVAL	The system call returns the value of **semval** for the *semnum*-th semaphore of the set. The calling process must have read permission on the semaphore set.
GETZCNT	The system call returns the value of **semzcnt** (i.e., the number of processes waiting for the value of this semaphore to become zero) for the *semnum*-th semaphore of the set (i.e. the number of processes waiting for **semval** of the *semnum*-th semaphore of the set to become 0). The calling process must have read permission on the semaphore set.
SETALL	Set **semval** for all semaphores of the set using *arg.array***,** updating also the*sem_ctime* member of the *semid_ds* structure associated with the set. Undo entries (see **semop**(2)) are cleared for altered semaphores in all processes. If the changes to semaphore values would permit blocked **semop**() calls in other processes to

	proceed, then those processes are woken up. The argument *semnum* is ignored. The calling process must have alter (write) permission on the semaphore set.
SETVAL	Set the value of **semval** to *arg.val* for the *semnum*-th semaphore of the set, updating also the *sem_ctime* member of the *semid_ds* structure associated with the set. Undo entries are cleared for altered semaphores in all processes. If the changes to semaphore values would permit blocked **semop**() calls in other processes to proceed, then those processes are woken up. The calling process must have alter permission on the semaphore set.

RETURN VALUE

On failure **semctl**() returns -1 with *errno* indicating the error.

NAME

semop - semaphore operations

SYNOPSIS

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

#include <sys/ipc.h>

#include <sys/sem.h>


intsemop(int semid, structsembuf *sops, unsigned nsops);
```

DESCRIPTION

Each semaphore in a semaphore set has the following associated values:

```
unsigned short  semval;   /* semaphore value */
unsigned short  semzcnt;  /* # waiting for zero */
unsigned short  semncnt;  /* # waiting for increase */
pid_tsempid;   /* process that did last op */
```

semop() performs operations on selected semaphores in the set indicated by *semid*. Each of the *nsops* elements in the array pointed to by *sops* specifies an operation to be performed on a single semaphore. The elements of this structure are of type *structsembuf*, containing the following members:

```
unsigned short sem_num;  /* semaphore number */
short          sem_op;   /* semaphore operation */
short          sem_flg;  /* operation flags */
```

Flags recognized in *sem_flg* are **IPC_NOWAIT** and **SEM_UNDO**. If an operation specifies **SEM_UNDO**, it will be automatically undone when the process terminates.

The set of operations contained in *sops* is performed *atomically*, that is, the operations are performed at the same time, and only if they can all be simultaneously performed. The behaviour of the system call if not all operations can be performed immediately depends on the presence of the **IPC_NOWAIT** flag in the individual *sem_flg* fields, as noted below.

Each operation is performed on the *sem_num*-th semaphore of the semaphore set, where the first semaphore of the set is numbered 0. There are three types of operation, distinguished by the value of *sem_op*.

If *sem_op* is a positive integer, the operation adds this value to the semaphore value (*semval*). Furthermore, if**SEM_UNDO** is specified for this operation, the system updates the process undo count (*semadj*) for this semaphore. This operation can always proceed — it never forces a process to wait. The calling process must have alter permission on the semaphore set.

If *sem_op* is zero, the process must have read permission on the semaphore set. This is a "wait-for-zero" operation: if *semval* is zero, the operation can immediately proceed. Otherwise, if **IPC_NOWAIT** is specified in *sem_flg*, **semop**() fails with *errno* set to **EAGAIN** (and none of the operations in *sops* is performed). Otherwise *semzcnt* (the count of processes waiting until this semaphore's value becomes zero) is incremented by one and the process sleeps until one of the following occurs:

Tag	Description

<table>
<tr><td>o</td><td><i>semval</i> becomes 0, at which time the value of <i>semzcnt</i> is decremented.</td></tr>
<tr><td>o</td><td>The semaphore set is removed: <b>semop</b>() fails, with <i>errno</i> set to <b>EIDRM</b>.</td></tr>
<tr><td>o</td><td>The calling process catches a signal: the value of <i>semzcnt</i> is decremented and <b>semop</b>() fails, with <i>errno</i> set to <b>EINTR</b>.</td></tr>
<tr><td>o</td><td>The time limit specified by <i>timeout</i> in a <b>semtimedop</b>() call expires: <b>semop</b>() fails, with <i>errno</i> set to <b>EAGAIN</b>.</td></tr>
<tr><td colspan="2">If <i>sem_op</i> is less than zero, the process must have alter permission on the semaphore set. If <i>semval</i> is greater than or equal to the absolute value of <i>sem_op</i>, the operation can proceed immediately: the absolute value of <i>sem_op</i> is subtracted from <i>semval</i>, and, if <b>SEM_UNDO</b> is specified for this operation, the system updates the process undo count (<i>semadj</i>) for this semaphore. If the absolute value of <i>sem_op</i> is greater than <i>semval</i>, and <b>IPC_NOWAIT</b> is specified in <i>sem_flg</i>, <b>semop</b>() fails, with <i>errno</i> set to <b>EAGAIN</b> (and none of the operations in <i>sops</i> is performed). Otherwise <i>semncnt</i> (the counter of processes waiting for this semaphore’s value to increase) is incremented by one and the process sleeps until one of the following occurs:</td></tr>
<tr><td>o</td><td><i>semval</i> becomes greater than or equal to the absolute value of <i>sem_op</i>, at which time the value of <i>semncnt</i> is decremented, the absolute value of <i>sem_op</i> is subtracted from<i>semval</i> and, if <b>SEM_UNDO</b> is specified for this operation, the system updates the process undo count (<i>semadj</i>) for this semaphore.</td></tr>
<tr><td>o</td><td>The semaphore set is removed from the system: <b>semop</b>() fails, with <i>errno</i> set to<b>EIDRM</b>.</td></tr>
<tr><td>o</td><td>The calling process catches a signal: the value of <i>semncnt</i> is decremented and<b>semop</b>() fails, with <i>errno</i> set to <b>EINTR</b>.</td></tr>
</table>

On successful completion, the *sempid* value for each semaphore specified in the array pointed to by *sops* is set to the process ID of the calling process. In addition, the *sem_otime* is set to the current time.

RETURN VALUE

If successful **semop**() return 0; otherwise they return -1 with *errno* indicating the error.

Unit-IV- Shared Memory, Multithreaded Programming

Shared Memory

Shared memory is the second of the three IPC facilities. It allows two unrelated processes to access the same logical memory. Shared memory is a very efficient way of transferring data between two running processes. Although the X/Open standard doesn't require it, it's probable that most implementations of shared memory arrange for the memory being shared between different processes to be the same physical memory.

Overview

Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process. Other processes can then 'attach' the same shared memory segment into their own address space. All processes can access the memory locations just as though the memory had been allocated by malloc. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory.

By itself, shared memory doesn't provide any synchronization facilities. There are no automatic facilities to prevent a second process starting to read the shared memory before the first process has finished writing to it. It's the responsibility of the programmer to synchronize access.

UNIX System API's for Shared Memory(shmget,shmat,shmdt,shmctl)

NAME

(i) shmget - allocates a shared memory segment

SYNOPSIS

#include <sys/ipc.h>

#include <sys/shm.h>

intshmget(key_t *key*, **size_t** *size*, **int** *shmflg***);**

DESCRIPTION

shmget() returns the identifier of the shared memory segment associated with the value of the argument *key*. A new shared memory segment, with size equal to the value of *size* rounded up to a multiple of **PAGE_SIZE**, is created if*key* has the value **IPC_PRIVATE** or *key* isn't **IPC_PRIVATE**, no shared memory segment corresponding to *key*exists, and **IPC_CREAT** is specified in *shmflg*.

If *shmflg* specifies both **IPC_CREAT** and **IPC_EXCL** and a shared memory segment already exists for *key*, then**shmget**() fails with *errno* set to **EEXIST**. (This is analogous to the effect of the combination **O_CREAT | O_EXCL** for **open**(2).)

The value *shmflg* is composed of:

Tag	Description
SHM_HUGETLB	used for allocating HUGETLB pages for shared memory. **IPC_CREAT** to create a new segment. If this flag is not used, then **shmget**() will find the segment associated with *key* and check to see if the user has permission to access the segment.
IPC_EXCL	used with **IPC_CREAT** to ensure failure if the segment already exists.
mode_flags	(least significant 9 bits) specifying the permissions granted to the owner, group, and world. These bits have the same format, and the same meaning, as the *mode*argument of **open**(2). Presently, the execute permissions are not used by the system.
SHM_HUGETLB (since Linux 2.6)	
	Allocate the segment using "huge pages." See the kernel source file*Documentation/vm/hugetlbpage.txt* for further information.
SHM_NORESERVE (since Linux 2.6.15)	
	This flag serves the same purpose as the **mmap**(2) **MAP_NORESERVE** flag. Do not reserve swap space for this segment. When swap space is reserved, one has the guarantee that it is possible to modify the segment. When swap space is not reserved one might get SIGSEGV upon a write if no physical memory is available. See also the discussion of the file */proc/sys/vm/overcommit_memory* in **proc**(5).

When a new shared memory segment is created, its contents are initialised to zero values, and its associated data structure, *shmid_ds* (see **shmctl**(2)), is initialised as follows:

Tag	Description
	shm_perm.cuid and *shm_perm.uid* are set to the effective user ID of the calling process.
	shm_perm.cgid and *shm_perm.gid* are set to the effective group ID of the calling process.
	The least significant 9 bits of *shm_perm.mode* are set to the least significant 9 bit of *shmflg*.

	shm_segsz is set to the value of *size*.
	shm_lpid, *shm_nattch*, *shm_atime* and *shm_dtime* are set to 0.
	shm_ctime is set to the current time.

If the shared memory segment already exists, the permissions are verified, and a check is made to see if it is marked for destruction.

RETURN VALUE

A valid segment identifier, *shmid*, is returned on success, -1 on error.

ERRORS

On failure, *errno* is set to one of the following:

Tag	Description
EACCES	The user does not have permission to access the shared memory segment, and does not have the **CAP_IPC_OWNER** capability.
EEXIST	**IPC_CREAT | IPC_EXCL** was specified and the segment exists.
EINVAL	A new segment was to be created and *size* < **SHMMIN** or *size* > **SHMMAX**, or no new segment was to be created, a segment with given key existed, but *size* is greater than the size of that segment.
ENFILE	The system limit on the total number of open files has been reached.
ENOENT	No segment exists for the given *key*, and **IPC_CREAT** was not specified.
ENOMEM	No memory could be allocated for segment overhead.
ENOSPC	All possible shared memory IDs have been taken (**SHMMNI**), or allocating a segment of the requested *size* would cause the system to exceed the system-wide limit on shared memory (**SHMALL**).
EPERM	The **SHM_HUGETLB** flag was specified, but the caller was not privileged (did not have the **CAP_IPC_LOCK** capability).

NOTES

IPC_PRIVATE isn't a flag field but a *key_t* type. If this special value is used for *key*, the system call ignores everything but the least significant 9 bits of *shmflg* and creates a new shared memory segment (on success).

The following limits on shared memory segment resources affect the **shmget**() call:

Tag	Description
SHMALL	System wide maximum of shared memory pages (on Linux, this limit can be read and modified via */proc/sys/kernel/shmall*).
SHMMAX	Maximum size in bytes for a shared memory segment: policy dependent (on Linux, this limit can be read and modified via */proc/sys/kernel/shmmax*).
SHMMIN	Minimum size in bytes for a shared memory segment: implementation dependent (currently 1 byte, though **PAGE_SIZE** is the effective minimum size).
SHMMNI	System wide maximum number of shared memory segments: implementation dependent (currently 4096, was 128 before Linux 2.3.99; on Linux, this limit can be read and modified via */proc/sys/kernel/shmmni*).

The implementation has no specific limits for the per process maximum number of shared memory segments (**SHMSEG**).

NAME

(ii) shmop - shared memory operations

SYNOPSIS

#include <sys/types.h>

#include <sys/shm.h>

void *shmat(int *shmid*, **const void** **shmaddr*, **int** *shmflg***);**

intshmdt(const void **shmaddr***);**

DESCRIPTION

shmat() attaches the shared memory segment identified by *shmid* to the address space of the calling process. The attaching address is specified by *shmaddr* with one of the following criteria:

If *shmaddr* is NULL, the system chooses a suitable (unused) address at which to attach the segment.

If *shmaddr* isn't NULL and **SHM_RND** is specified in *shmflg*, the attach occurs at the address equal to *shmaddr*rounded down to the nearest multiple of **SHMLBA**. Otherwise *shmaddr* must be a page-aligned address at which the attach occurs.

If **SHM_RDONLY** is specified in *shmflg*, the segment is attached for reading and the process must have read permission for the segment. Otherwise the segment is attached for read and write and the process must have read and write permission for the segment. There is no notion of a write-only shared memory segment.

The (Linux-specific) **SHM_REMAP** flag may be specified in *shmflg* to indicate that the mapping of the segment should replace any existing mapping in the range starting at *shmaddr* and continuing for the size of the segment. (Normally an **EINVAL** error would result if a mapping already exists in this address range.) In this case, *shmaddr*must not be NULL.

The **brk**(2) value of the calling process is not altered by the attach. The segment will automatically be detached at process exit. The same segment may be attached as a read and as a read-write one, and more than once, in the process's address space.

A successful **shmat**() call updates the members of the **shmid_ds** structure (see **shmctl**(2)) associated with the shared memory segment as follows:

<table>
<tr><th>Tag</th><th>Description</th></tr>
<tr><td></td><td><i>shm_atime</i> is set to the current time.</td></tr>
<tr><td></td><td><i>shm_lpid</i> is set to the process-ID of the calling process.</td></tr>
<tr><td></td><td><i>shm_nattch</i> is incremented by one.</td></tr>
<tr><td colspan="2"><b>shmdt</b>() detaches the shared memory segment located at the address specified by <i>shmaddr</i> from the address space of the calling process. The to-be-detached segment must be currently attached with <i>shmaddr</i> equal to the value returned by the attaching <b>shmat</b>() call.</td></tr>
<tr><td colspan="2">On a successful <b>shmdt</b>() call the system updates the members of the <i>shmid_ds</i> structure associated with the shared memory segment as follows:</td></tr>
<tr><td></td><td><i>shm_dtime</i> is set to the current time.</td></tr>
<tr><td></td><td><i>shm_lpid</i> is set to the process-ID of the calling process.</td></tr>
<tr><td></td><td><i>shm_nattch</i> is decremented by one. If it becomes 0 and the segment is marked for deletion, the segment is deleted.</td></tr>
</table>

RETURN VALUE

On success **shmat**() returns the address of the attached shared memory segment; on error *(void *) -1* is returned, and*errno* is set to indicate the cause of the error.

On success **shmdt**() returns 0; on error -1 is returned, and *errno* is set to indicate the cause of the error.

ERRORS

When **shmat**() fails, *errno* is set to one of the following:

Tag	Description
EACCES	The calling process does not have the required permissions for the requested attach type, and does not have the **CAP_IPC_OWNER** capability.
EINVAL	Invalid *shmid* value, unaligned (i.e., not page-aligned and **SHM_RND** was not specified) or invalid *shmaddr* value, or failing attach at **brk**(), or **SHM_REMAP**was specified and *shmaddr* was NULL.
ENOMEM	Could not allocate memory for the descriptor or for the page tables.

When **shmdt**() fails, *errno* is set as follows:

Tag	Description
EINVAL	There is no shared memory segment attached at *shmaddr*; or, *shmaddr* is not aligned on a page boundary.

NAME

(iii) shmctl - shared memory control

SYNOPSIS

#include <sys/ipc.h> #include <sys/shm.h>

intshmctl(int *shmid*, **int** *cmd*, **structshmid_ds** ***buf*);

DESCRIPTION

shmctl() performs the control operation specified by *cmd* on the shared memory segment whose identifier is given in *shmid*.

The *buf* argument is a pointer to a *shmid_ds* structure, defined in <sys/shm.h> as follows:

```
structshmid_ds {
structipc_permshm_perm;    /* Ownership and permissions */
size_tshm_segsz;   /* Size of segment (bytes) */
time_tshm_atime;   /* Last attach time */
time_tshm_dtime;   /* Last detach time */
time_tshm_ctime;   /* Last change time */
pid_tshm_cpid;    /* PID of creator */
pid_tshm_lpid;    /* PID of last shmat()/shmdt() */
shmatt_tshm_nattch;  /* No. of current attaches */
    ...
};
```

The *ipc_perm* structure is defined in <sys/ipc.h> as follows (the highlighted fields are settable using **IPC_SET**):

```
structipc_perm {
key_t key;            /* Key supplied to shmget() */
uid_tuid;             /* Effective UID of owner */
gid_tgid;             /* Effective GID of owner */
uid_tcuid;            /* Effective UID of creator */
gid_tcgid;            /* Effective GID of creator */
    unsigned short mode;  /* Permissions + SHM_DEST and
                             SHM_LOCKED flags */
    unsigned short seq;   /* Sequence number */
};
```

Valid values for *cmd* are:

Tag	Description
IPC_STAT	Copy information from the kernel data structure associated with *shmid* into the*shmid_ds* structure pointed to by *buf*. The caller must have read permission on the shared memory segment.
IPC_SET	Write the values of some members of the *shmid_ds* structure pointed to by *arg.buf* to the kernel data structure associated with this shared memory segment, updating also its *shm_ctime* member. The following fields can be changed: *shm_perm.uid*, *shm_perm.gid*, and (the least significant 9 bits of) *shm_perm.mode*. The effective UID of the calling process must match the owner (*shm_perm.uid*) or creator (*shm_perm.cuid*) of the shared memory segment, or the caller must be privileged.
IPC_RMID	Mark the segment to be destroyed. The segment will only actually be destroyed after the last process detaches it (i.e., when the *shm_nattch* member of the associated structure *shmid_ds* is zero). The caller must be the owner or creator, or be privileged. If a segment has been marked for destruction, then the (non-standard) **SHM_DEST**flag of the *shm_perm.mode* field in the

	associated data structure retrieved by**IPC_STAT** will be set.

RETURN VALUE

ERRORS

Tag	Description
EACCES	**IPC_STAT** or **SHM_STAT** is requested and *shm_perm.mode* does not allow read access for *shmid*, and the calling process does not have the **CAP_IPC_OWNER**capability.
EFAULT	The argument *cmd* has value **IPC_SET** or **IPC_STAT** but the address pointed to by*buf* isn't accessible.
EIDRM	*shmid* points to a removed identifier.
EINVAL	*shmid* is not a valid identifier, or *cmd* is not a valid command. Or: for a **SHM_STAT**operation, the index value specified in *shmid* referred to an array slot that is currently unused.

Thread Basics:

- Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
  - Process instructions
  - Most data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - User and group id
- Each thread has a unique:
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses

    - signal mask
    - priority
    - Return value: errno
- pthread functions return "0" if OK.

POSIX Thread API’s:

-

```
intpthread_create(pthread_t * thread,
constpthread_attr_t * attr,
                    void * (*start_routine)(void *),
void *arg);
```

- Arguments:
    - thread - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
    - attr - Set to NULL if default thread attributes are used. (else define members of the struct pthread_attr_t defined in bits/pthreadtypes.h) Attributes include:
        - detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
        - scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
        - scheduling parameter
        - inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
        - scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
        - guard size
        - stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
        - stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),
    - void * (*start_routine) - pointer to the function to be threaded. Function has a single argument: pointer to void.
    - *arg - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.
- Function call: **pthread_exit**

```
void pthread_exit(void *retval);
```

Arguments:

    - retval - Return value of thread.

This routine kills the thread. The pthread_exit function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join.

Thread Management

Joining and Detaching Threads

Routines:

pthread_join (threadid,status)

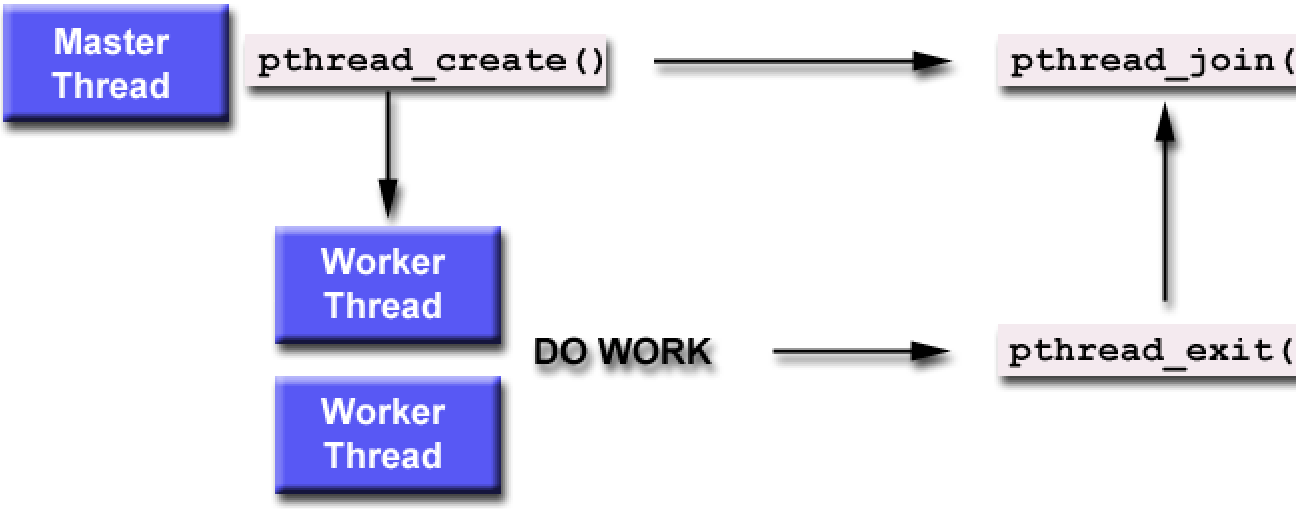
pthread_detach (threadid)

pthread_attr_setdetachstate (attr,detachstate)

pthread_attr_getdetachstate (attr,detachstate)

Joining:

- "Joining" is one way to accomplish synchronization between threads. For example:



- The **pthread_join()** subroutine blocks the calling thread until the specified **threadid** thread terminates.
- The programmer is able to obtain the target thread's termination return **status** if it was specified in the target thread's call to **pthread_exit()**.
- A joining thread can match one **pthread_join()** call. It is a logical error to attempt multiple joins on the same thread.
- Two other synchronization methods, mutexes and condition variables, will be discussed later.

Joinable or Not?

- When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable.
- To explicitly create a thread as joinable or detached, the **attr** argument in the **pthread_create**() routine is used. The typical 4 step process is:
    1. Declare a pthread attribute variable of the **pthread_attr_t** data type
    2. Initialize the attribute variable with **pthread_attr_init()**
    3. Set the attribute detached status with **pthread_attr_setdetachstate()**
    4. When done, free library resources used by the attribute with **pthread_attr_destroy()**

Detaching:

- The **pthread_detach()** routine can be used to explicitly detach a thread even though it was created as joinable.
- There is no converse routine.

Thread Management

Joining and Detaching Threads

Routines:

pthread_join (threadid,status)

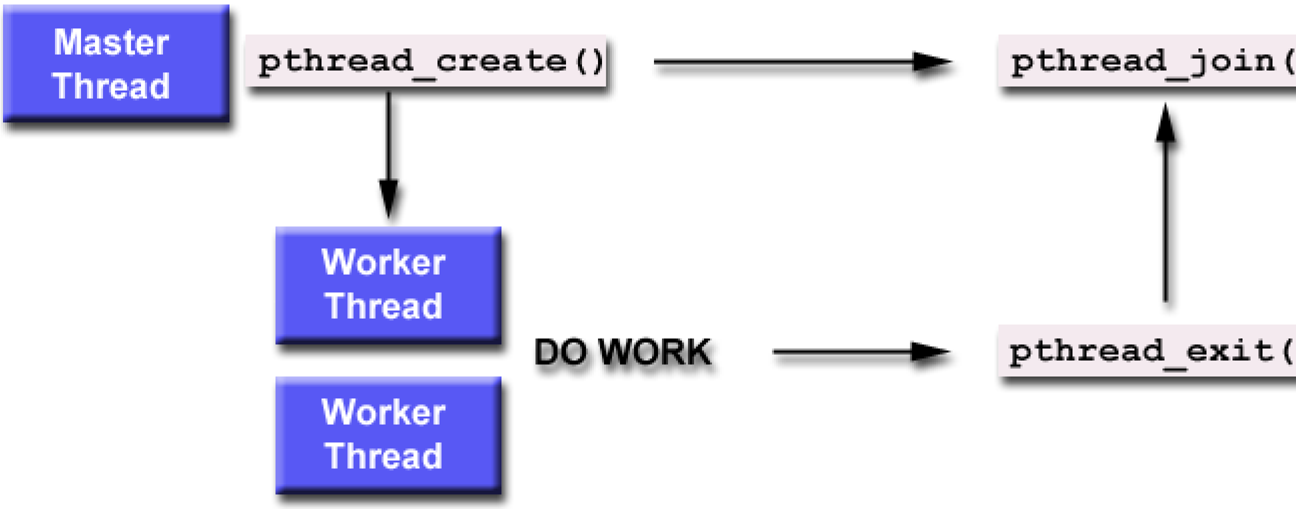
pthread_detach (threadid)

pthread_attr_setdetachstate (attr,detachstate)

pthread_attr_getdetachstate (attr,detachstate)

Joining:

- "Joining" is one way to accomplish synchronization between threads. For example:



- The **pthread_join()** subroutine blocks the calling thread until the specified **threadid** thread terminates.
- The programmer is able to obtain the target thread's termination return **status** if it was specified in the target thread's call to **pthread_exit**().
- A joining thread can match one **pthread_join**() call. It is a logical error to attempt multiple joins on the same thread.
- Two other synchronization methods, mutexes and condition variables, will be discussed later.

Joinable or Not?

- When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable.
- To explicitly create a thread as joinable or detached, the **attr** argument in the **pthread_create**() routine is used. The typical 4 step process is:
    1. Declare a pthread attribute variable of the **pthread_attr_t** data type
    2. Initialize the attribute variable with **pthread_attr_init**()
    3. Set the attribute detached status with **pthread_attr_setdetachstate**()
    4. When done, free library resources used by the attribute with **pthread_attr_destroy**()

Detaching:

- The **pthread_detach**() routine can be used to explicitly detach a thread even though it was created as joinable.
- There is no converse routine.

Process:

- An executing instance of a program is called a process.
- Some operating systems use the term ‘task‘ to refer to a program that is being executed.

- A process is always stored in the main memory also termed as the primary memory or random access memory.
- Therefore, a process is termed as an active entity. It disappears if the machine is rebooted.
- Several process may be associated with a same program.
- On a multiprocessor system, multiple processes can be executed in parallel.
- On a uni-processor system, though true parallelism is not achieved, a process scheduling algorithm is applied and the processor is scheduled to execute each process one at a time yielding an illusion of concurrency.
- **Example:** Executing multiple instances of the 'Calculator' program. Each of the instances are termed as a process.

Thread:

- A thread is a subset of the process.
- It is termed as a 'lightweight process', since it is similar to a real process but executes within the context of a process and shares the same resources allotted to the process by the kernel.
- Usually, a process has only one thread of control – one set of machine instructions executing at a time.
- A process may also be made up of multiple threads of execution that execute instructions concurrently.
- Multiple threads of control can exploit the true parallelism possible on multiprocessor systems.
- On a uni-processor system, a thread scheduling algorithm is applied and the processor is scheduled to run each thread one at a time.
- All the threads running within a process share the same address space, file descriptors, stack and other process related attributes.
- Since the threads of a process share the same memory, synchronizing the access to the shared data withing the process gains unprecedented importance.

What is the difference between threads and processes?

The major differences between threads and processes are:

1. Threads share the address space of the process that created it; processes have their own address space.
2. Threads have direct access to the data segment of its process; processes have their own copy of the data segment of the parent process.
3. Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.
4. Threads have almost no overhead; processes have considerable overhead.
5. New threads are easily created; new processes require duplication of the parent process.

6. Threads can exercise considerable control over threads of the same process; processes can only exercise control over child processes.
7. Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process; changes to the parent process do not affect child processes.

8. Comparison between Process and Thread:

	Process	**Thread**
Definition	An executing instance of a program is called a process.	A thread is a subset of the process.
Process	It has its own copy of the data segment of the parent process.	It has direct access to the data segment of its process.
Communication	Processes must use inter-process communication to communicate with sibling processes.	Threads can directly communicate with other threads of its process.
Overheads	Processes have considerable overhead.	Threads have almost no overhead.
Creation	New processes require duplication of the parent process.	New threads are easily created.
Control	Processes can only exercise control over child processes.	Threads can exercise considerable control over threads of the same process.
Changes	Any change in the parent process does not affect child processes.	Any change in the main thread may affect the behavior of the other threads of the process.
Memory	Run in separate memory spaces.	Run in shared memory spaces.

File descriptors	Most file descriptors are not shared.	It shares file descriptors.
File system	There is no sharing of file system context.	It shares file system context.
Signal	It does not share signal handling.	It shares signal handling.
Controlled by	Process is controlled by the operating system.	Threads are controlled by programmer in a program.
Dependence	Processes are independent.	Threads are dependent.

Basic Thread Functions: Creation and Termination

In this section, we will cover five basic thread functions and then use these in the next two sections to recode o

pthread_createFunction

When a program is started by exec, a single thread is created, called the initial thread or main thread. Addition

```
#include <pthread.h>
intpthread_create(pthread_t *tid, constpthread_attr_t *attr, void *(*func) (void *), void *arg);
```

Returns: 0 if OK, positive Exxx value on error

Each thread within a process is identified by a thread ID, whose datatype is pthread_t(often an unsigned int). C

Each thread has numerous attributes: its priority, its initial stack size, whether it should be a daemon thread or overrides the default. We normally take the default, in which case, we specify the attr argument as a null point

Finally, when we create a thread, we specify a function for it to execute. The thread starts by calling this functi address of the function is specified as the func argument, and this function is called with a single pointer argum of this structure as the single argument to the start function.

Notice the declarations of func and arg. The function takes one argument, a generic pointer (void *), and retur one pointer (again, to anything we want).

The return value from the Pthread functions is normally 0 if successful or nonzero on an error. But unlike the s

return the positive error indication as the function's return value. For example, if pthread_createcannot create a
The Pthread functions do not set errno. The convention of 0 for success or nonzero for an error is fine since all

pthread_joinFunction

We can wait for a given thread to terminate by calling pthread_join. Comparing threads to Unix processes, pth

```
#include <pthread.h>
intpthread_join (pthread_ttid, void ** status);
```

Returns: 0 if OK, positive Exxx value on error

We must specify the tid of the thread that we want to wait for. Unfortunately, there is no way to wait for any o
(similar towaitpidwith a process ID argument of 1). We will return to this problem when we discuss Figure 26.

If the status pointer is non-null, the return value from the thread (a pointer to some object) is stored in the locat
by status.

pthread_selfFunction

Each thread has an ID that identifies it within a given process. The thread ID is returned by pthread_createand
used by pthread_join. A thread fetches this value for itself using pthread_self.

```
#include <pthread.h>
pthread_tpthread_self (void);
```

Returns: thread ID of calling thread

Comparing threads to Unix processes, pthread_selfis similar togetpid. pthread_detachFunction

A thread is either joinable (the default) or detached. When a joinable thread terminates, its thread ID and exit s
retained until another thread calls pthread_join. But a detached thread is like a daemon process: When it termin
resources are released and we cannot wait for it to terminate. If one thread needs to know when another thread
is best to leave the thread as joinable.

The pthread_detachfunction changes the specified thread so that it is detached.

```
#include <pthread.h>
intpthread_detach (pthread_ttid);
```

Returns: 0 if OK, positive Exxx value on error

This function is commonly called by the thread that wants to detach itself, as in

```
pthread_detach (pthread_self());
```

pthread_exitFunction One way for a thread to terminate is to call pthread_exit.

```
#include <pthread.h>
void pthread_exit (void *status);
```

Does not return to caller

If the thread is not detached, its thread ID and exit status are retained for a later pthread_joinby some other thre calling process.

The pointer status must not point to an object that is local to the calling thread since that object disappears whe thread terminates.

There are two other ways for a thread to terminate:

□ The function that started the thread (the third argument to pthread_create) can return. Since tl must be declared as returning a void pointer, that return value is the exit status of the thread.

□ If the main function of the process returns or if any thread calls exit, the process terminates, i any threads.

Synchronization

We saw that both threads are executing together, but our method of switching between them was clumsy and inefficient. Fortunately, there are a set functions specifically for giving us better ways to control the execution of threads and access to critical sections of code.

We will look at two basic methods here. The first is **semaphores**, which act like gatekeepers around a piece of code. The second is **mutexes**, which act as a mutual exclusion (hence mutex) device to protect sections of code.

Both are similar. (Indeed, one can be implemented in terms of the other.) However, there are some cases where the semantics of the problem suggest one is more expressive than the other. For example, controlling access to some shared memory, which only one thread can access it at a time, would most naturally involve a mutex. However, controlling access to a set of identical objects as a whole, such as giving a telephone line to a thread out of a set of five available lines, suits a counting semaphore better. Which one you choose depends on personal preference and the most appropriate mechanism for your program.

Synchronization with Semaphores

Important There are two sets of interface functions for semaphores. One is taken from POSIX Realtime Extensions and used for threads. The other is known as System V semaphores, which are commonly used for process synchronization. (We will we will meet the second type in a later chapter.) The two are not guaranteed interchangeable and, although very similar, use different function calls.

We are going to look at the simplest type of semaphore, a binary semaphore that takes only values 0 or 1. There is a more general semaphore, a counting semaphore that takes a wider range of values. Normally semaphores are used to protect a piece of code so that only one thread of execution can run it at any one time and, for this job, a binary semaphore is needed. Occasionally, we want to permit a limited number of threads to execute a given piece of code and, for this, we would use a counting semaphore. Since counting semaphores are much less common, we won't consider them further here, except to say that they are just a logical extension of a binary semaphore and that the actual function calls needed are identical.

The semaphore functions do not start with pthread_, like most thread specific functions but with sem_. There are four basic semaphore functions used in threads. They are all quite simple.

POSIX API's for Semaphores: (sem_init,sem_wait,sem_post,sem_destroy)

A semaphore is created with the sem_init function, which is declared as follows.

```
#include <semaphore.h>
intsem_init(sem_t *sem, intpshared, unsigned int value);
```

This function initializes a semaphore object pointed to by sem, sets its sharing option (of which more in a moment), and gives it an initial integer value. The pshared parameter controls the type of semaphore. If the value of pshared is 0, then the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes. Here we are only interested in semaphores that are not shared between processes. At the time of writing Linux doesn't support this sharing, and passing a non−zero value for pshared will cause the call to fail.

The next pair of functions control the value of the semaphore and are declared as follows.

```
#include <semaphore.h>
intsem_wait(sem_t * sem);
intsem_post(sem_t * sem);
```

These both take a pointer to the semaphore object initialized by a call to sem_init.

The sem_post function atomically increases the value of the semaphore by 1. "**Atomically**" here means that, if two threads simultaneously try and increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment and write a value to a file at the same time. The semaphore will always be correctly increased in value by 2, since two threads tried to change it.

The sem_wait function atomically decreases the value of the semaphore by one, but always waits till the semaphore has a non−zero count first. Thus if you call sem_wait on a semaphore with a value of 2, the thread will continue executing but the semaphore will be decreased to 1. If sem_wait is called on a semaphore with a value of 0, then the function will wait until some other thread has incremented the value so that it is no longer 0. If two threads are both waiting in sem_wait for the same semaphore to become non−zero and it is incremented once by a third process, then only one of the two waiting processes will get to decrement the semaphore and continue; the other will remain waiting.

This atomic 'test and set' ability in a single function is what makes semaphores so valuable. There is another semaphore function, sem_trywait that is the non−blocking partner of sem_wait. We don't discuss it further here, but you can find more details in the manual pages.

The last semaphore function is sem_destroy. This function tidies up the semaphore when we have finished with it. It is declared as follows.

```
#include <semaphore.h>
intsem_destroy(sem_t * sem);
```

Again, this function takes a pointer to a semaphore and tidies up any resources that it may have. If you attempt to destroy a semaphore for which some thread is waiting, you will get an error.

Like most Linux functions, these functions all return 0 on success.

Synchronization with Mutexes

The other way of synchronizing access in multithreaded programs is with **mutexes**. These act by allowing the programmer to 'lock' an object, so that only one thread can access it. To control access to a critical section of code you lock a mutex before entering the code section and then unlock it when you have finished.

The basic functions required to use mutexes are very similar to those needed for semaphores. They are declared as follows.

```
#include <pthread.h>
intpthread_mutex_init(pthread_mutex_t *mutex, constpthread_mutexattr_t *mutexattr);
intpthread_mutex_lock(pthread_mutex_t *mutex));
intpthread_mutex_unlock(pthread_mutex_t *mutex);
intpthread_mutex_destroy(pthread_mutex_t *mutex);
```

As usual, 0 is returned for success, on failure an error code is returned, but errno is not set, you must use the return code.

As with semaphores, they all take a pointer to a previously declared object, this time a pthread_mutex_t. The extra attribute parameter pthread_mutex_init allows us to provide attributes for the mutex, which control its behavior. The attribute type by default is 'fast'. This has the slight drawback that, if your program tries to call pthread_mutex_lock on a mutex that it already has locked, then the program will block. Since the thread that holds the lock is the one that is now blocked, the mutex can never be unlocked and the program is deadlocked. It is possible to alter the attributes of the mutex so that it either checks for this and returns an error or acts recursively and allows multiple locks by the same thread if there are the same number of unlocks afterwards.

Setting the attribute of a mutex is beyond the scope of this book, so we will pass NULL for the attribute pointer, and use the default behavior. You can find more about changing the attributes in the manual page for pthread_mutex_init.

Thread Attributes

When we first looked at threads, we did not discuss the question of thread attributes. We will now do so. There are quite a few attributes of threads that you can control. However, here we are only going to look at those that you are most likely to need. Details of the others can be found in the manual pages.

In all our previous examples, we have had to re–synchronize our threads using pthread_join before we allow the program to exit. We need to do this if we want to allow one thread to return data to the thread that created it. Sometimes, we neither need the second thread to return information to the main thread nor want the main thread to wait for it.

Suppose that we create a second thread to spool a backup copy of a data file that is being edited while the main thread continues to service the user. When the backup has finished the second thread can just terminate. There is no need for it to re–join the main thread.

We can create threads that behave like this. They are called **detached threads**, and we create them by modifying the thread attributes or by calling pthread_detach. Since we want to demonstrate attributes, we will use the former method here.

The most important function that we need is pthread_attr_init, which initializes a thread attribute object.

```
#include <pthread.h>
intpthread_attr_init(pthread_attr_t *attr);
```

Once again, 0 is returned for success, and an error code is returned on failure.

There is also a destroy function, pthread_attr_destroy, but at the time of writing its implementation in Linux is to do nothing. Its purpose is to allow clean destruction of the attribute object, and you should call it, even though it currently does nothing on Linux, just in case the implementation one day changes and requires it to be called.

When we have a thread attribute object initialized, there are many additional functions that we can call to set different attribute behaviors. We will list them all here but look closely at only two. Here is the list of attribute functions that can be used.

```
intpthread_attr_setdetachstate(pthread_attr_t *attr, intdetachstate);
intpthread_attr_getdetachstate(constpthread_attr_t *attr, int *detachstate);
intpthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
intpthread_attr_getschedpolicy(constpthread_attr_t *attr, int *policy);
intpthread_attr_setschedparam(pthread_attr_t *attr, conststructsched_param *param);
intpthread_attr_getschedparam(constpthread_attr_t *attr, structsched_param *param);
intpthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
intpthread_attr_getinheritsched(constpthread_attr_t *attr, int *inherit);
intpthread_attr_setscope(pthread_attr_t *attr, int scope);
intpthread_attr_getscope(constpthread_attr_t *attr, int *scope);
intpthread_attr_setstacksize(pthread_attr_t *attr, int scope);
intpthread_attr_getstacksize(constpthread_attr_t *attr, int *scope);
```

As you can see, there are quite a lot of attributes.

detachedstate

This attribute allows us to avoid the need for threads to re–join. Like most of these _set functions, it takes a pointer to the attribute and a flag to determine the state required. The two possible flag values for pthread_attr_setdetachstate are PTHREAD_CREATE_JOINABLE and PTHREAD_CREATE_DETACHED. By default, the attribute will have value PTHREAD_CREATE_JOINABLE so that we should allow the two threads to join. If the state is set to PTHREAD_CREATE_DETACHED, then you cannot call pthread_join to recover the exit state of another thread.

schedpolicy

This controls how threads are scheduled. The options are SCHED_OTHER, SCHED_RP and SCHED_FIFO. By default, the attribute is SCHED_OTHER. The other two types of scheduling are only available to processes running with superuser permissions, as they both have real time scheduling but with slightly different behavior. SCHED_RR uses a round–robin scheduling scheme, and SCHED_FIFO uses a 'first in, first out' policy. Discussion of these is beyond the scope of this book.

schedparam

schedparam

This is a partner to schedpolicy and allows control over the scheduling of threads running with schedule policy SCHED_OTHER. We will have a look at an example of this in a short while.

inheritsched

This attribute takes two possible values, PTHREAD_EXPLICIT_SCHED and PTHREAD_INHERIT_SCHED. By default, the value is PTHREAD_EXPLICIT_SCHED, which means scheduling is explicitly set by the attributes. By setting it to PTHREAD_INHERIT_SCHED, a new thread will instead use the parameters that its creator thread was using.

scope

This attribute controls how scheduling of a thread is calculated. Since Linux only currently supports the value PTHREAD_SCOPE_SYSTEM, we will not look at this further here.

stacksize

This attribute controls the thread creation stack size, set in bytes. This is part of the 'optional' section of the specification and is only supported on implementations where _POSIX_THREAD_ATTR_STACKSIZE is defined. Linux implements threads with a large amount of stack by default, so the feature is generally redundant on Linux and consequently not implemented.

UNIT – V
SOCKETS:

3.1 Introduction

This chapter begins the description of the sockets API. We begin with socket address structures, which will be found in almost every example in the text. These structures can be passed in two directions: from the process to the kernel, and from the kernel to the process. The latter case is an example of a value-result argument, and we will encounter other examples of these arguments throughout the text.

The address conversion functions convert between a text representation of an address and the binary value that goes into a socket address structure. Most existing IPv4 code uses inet_addr and inet_ntoa, but two new functions, inet_pton and inet_ntop, handle both IPv4 and IPv6.

One problem with these address conversion functions is that they are dependent on the type of address being converted: IPv4 or IPv6. We will develop a set of functions whose names begin with sock_ that work with socket address structures in a protocol-independent fashion. We will use these throughout the text to make our code protocol-independent.

Generic Socket Address Structure A socket address structures is always passed by reference when passed as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from any of the supported protocol families. A problem arises in how to declare the type of pointer that is passed. With ANSI C, the solution is simple: void * is the generic pointer type. But, the socket functions predate ANSI C and the solution chosen in 1982 was to define a generic socket address structure in the <sys/socket.h>header, which we show in Figure 3.3. Figure 3.3 The generic socket address structure: sockaddr. struct sockaddr {

```
.  uint8_t      sa_len;
.  sa_family_t  sa_family;   /* address family: AF_xxx value */
.
.  char         sa_data[14];  /* protocol-specific address */
.  };
```

The socket functions are then defined as taking a pointer to the generic socket address structure, as shown here in the ANSI C function prototype for the bind function: int bind(int, struct sockaddr *, socklen_t);

. This requires that any calls to these functions must cast the pointer to the protocol-specific socket address structure to be a pointer to a generic socket address structure. For example, struct sockaddr_in serv; /* IPv4 socket address structure */

```
. /* fill in serv{} */
. bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

. If we omit the cast "(struct sockaddr *)," the C compiler generates a warning of the form "warning: passing arg 2 of 'bind' from incompatible pointer type," assuming the system's headers have an ANSI C prototype for the bind function.

IPv4 Socket Address Structure

An IPv4 socket address structure, commonly called an "Internet socket address structure," is named sockaddr_in and is defined by including the <netinet/in.h>header. Figure 3.1 shows the POSIX definition.

Figure 3.1 The Internet (IPv4) socket address structure: sockaddr_in. struct in_addr {

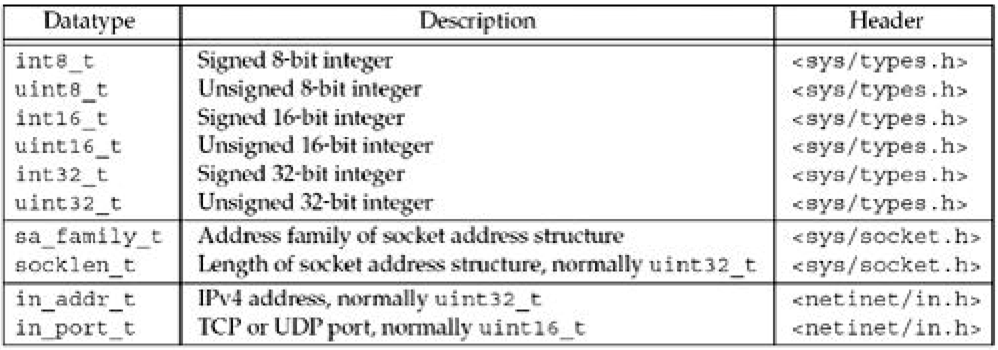
```
  in_addr_t   s_addr; /* 32-bit IPv4 address */
                    /* network byte ordered */
};

struct sockaddr_in {

uint8_t sin_len; /* length of structure (16) */

sa_family_t sin_family; /* AF_INET */

in_port_t sin_port; /* 16-bit TCP or UDP port number */
                    /* network byte ordered */

struct in_addr  sin_addr; /* 32-bit IPv4 address */
                         /* network byte ordered */

  char sin_zero[8];  /* unused */
};
```

There are several points we need to make about socket address structures in general using this example:

□ The length member, sin_len, was added with 4.3BSD-Reno, when support for the OSI protocols was added (Figure 1.15). Before this release, the first member was sin_family, which was historically an unsigned short. Not all vendors support a length field for socket address structures and the POSIX specification does not require this member. The datatype that we show, uint8_t, is typical, and POSIX-compliant systems provide datatypes of this form (Figure 3.2).

Figure 3.2. Datatypes required by the POSIX specification.

Having a length field simplifies the handling of variable-length socket address structures.



Datatype	Description	Header
int8_t	Signed 8-bit integer	<sys/types.h>
uint8_t	Unsigned 8-bit integer	<sys/types.h>
int16_t	Signed 16-bit integer	<sys/types.h>
uint16_t	Unsigned 16-bit integer	<sys/types.h>
int32_t	Signed 32-bit integer	<sys/types.h>
uint32_t	Unsigned 32-bit integer	<sys/types.h>
sa_family_t	Address family of socket address structure	<sys/socket.h>
socklen_t	Length of socket address structure, normally uint32_t	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32_t	<netinet/in.h>
in_port_t	TCP or UDP port, normally uint16_t	<netinet/in.h>

Even if the length field is present, we need never set it and need never examine it,

- unless we are dealing with routing sockets (Chapter 18). It is used within the kernel by the routines that deal with socket address structures from various protocol families (e.g., the routing table code).
- The four socket functions that pass a socket address structure from the process to the kernel, bind, connect, sendto, and sendmsg, all go through the sockargs function in a Berkeley-derived implementation (p. 452 of TCPv2). This function copies the socket address structure from the process and explicitly sets its sin_len member to the size of the structure that was passed as an argument to these four functions. The five socket functions that pass a socket address structure from the kernel to the process, accept, recvfrom, recvmsg, getpeername, and getsockname, all set the sin_len member before returning to the process.

- The POSIX specification requires only three members in the structure: sin_family, sin_addr, and sin_port. It is acceptable for a POSIX-compliant implementation to define additional structure members, and this is normal for an Internet socket address structure. Almost all implementations add the sin_zero member so that all socket address structures are at least 16 bytes in size.

- We show the POSIX datatypes for the s_addr, sin_family, and sin_port members. The in_addr_t datatype must be an unsigned integer type of at least 32 bits, in_port_t must be an unsigned integer type of at least 16 bits, and sa_family_t can be any unsigned integer type. The latter is normally an 8-bit unsigned integer if the implementation supports the length field, or an unsigned 16-bit integer if the length field is not supported. Figure 3.2 lists these three POSIX-defined datatypes, along with some other POSIX datatypes that we will encounter.

IPv6 Socket Address Structure

The IPv6 socket address is defined by including the <netinet/in.h>header, and we show it in Figure 3.4.

Figure 3.4 IPv6 socket address structure: sockaddr_in6.

```
struct in6_addr {
  uint8_t  s6_addr[16];         /* 128-bit IPv6 address */
```

```
                                  /* network byte ordered */
};
#define SIN6_LEN      /* required for compile-time tests */
struct sockaddr_in6 {

uint8_t
sa_family_t
in_port_t
                  sin6_len;
                  sin6_family;
                  sin6_port;
  uint32_t
  struct in6_addr sin6_addr;      /* IPv6 address */
                                  /* network byte ordered */
  uint32_t        sin6_scope_id; /* set of interfaces for a scope */
};
```

The extensions to the sockets API for IPv6 are defined in RFC 3493 [Gilligan et al. 2003]. Note the following points about Figure 3.4:

- The SIN6_LEN constant must be defined if the system supports the length member for socket address structures.
- The IPv6 family is AF_INET6, whereas the IPv4 family is AF_INET.
- The members in this structure are ordered so that if the sockaddr_in6 structure is 64-bit aligned, so is the 128-bit sin6_addr member. On some 64-bit processors, data accesses of 64-bit values are optimized if stored on a 64-bit boundary.
- The sin6_flowinfo member is divided into two fields:

o The low-order 20 bits are the flow label

o The high-order 12 bits are reserved

3.3 Value-Result Arguments

We mentioned that when a socket address structure is passed to any socket function, it is always passed by reference. That is, a pointer to the structure is passed. The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed: from the process to the kernel, or vice versa.

1. Three functions, bind, connect, and sendto, pass a socket address structure from the process to the kernel. One argument to these three functions is the pointer to the socket address structure and another argument is the integer size of the structure, as in

2. 3. 4. 5. struct sockaddr_in serv; 6.

```
. /* fill in serv{} */

. connect (sockfd, (SA *) &serv, sizeof(serv));
```

9.

Since the kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel. Figure 3.7 shows this scenario.

Figure 3.7. Socket address structure passed from process to kernel.

We will see in the next chapter that the datatype for the size of a socket address structure is actually socklen_t and not int, but the POSIX specification recommends that socklen_t be defined as uint32_t.

10. Four functions, accept, recvfrom, getsockname, and getpeername, pass a socket address structure from the kernel to the process, the reverse direction from the

previous scenario. Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure, as in

11. 12. 13.

```
. struct sockaddr_un cli; /* Unix domain */

. socklen_t len;
```

16.

```
. len = sizeof(cli); /* len is a value */

. getpeername(unixfd, (SA *) &cli, &len);

. /* len may have changed */
```

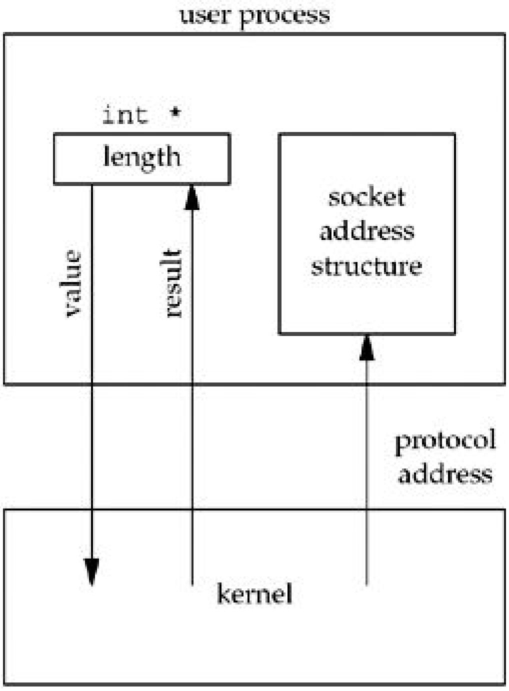
20.

The reason that the size changes from an integer to be a pointer to an integer is because the size is both a value when the function is called (it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in) and a result when the function returns (it tells the process how much information the kernel actually stored in the structure). This type of argument is called a value-result argument. Figure 3.8 shows this scenario.

Figure 3.8. Socket address structure passed from kernel to process.

We will see an example of value-result arguments in Figure 4.11.

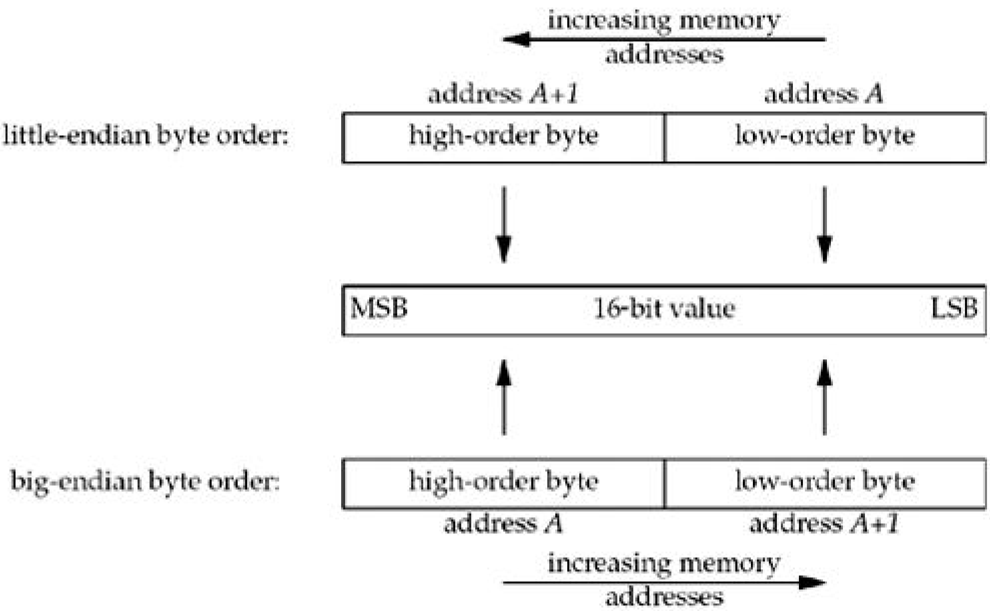
We have been talking about socket address structures being passed between the process and the kernel. For an implementation such as 4.4BSD, where all the socket functions are system calls within the kernel, this is correct. But in some implementations, notably System V, socket functions are just library functions that execute as part of a normal user process. How these functions interface with the protocol stack in the kernel is an implementation detail that normally does not affect us. Nevertheless, for simplicity, we will continue to talk about these structures as being passed between the process and the kernel by functions such as bind and connect. (We will see in Section C.1 that System V implementations do indeed pass socket address structures between processes and the kernel, but as part of STREAMS messages.)



3.4 Byte Ordering Functions

Consider a 16-bit integer that is made up of 2 bytes. There are two ways to store the two bytes in memory: with the low-order byte at the starting address, known as little-endian byte order, or with the high-order byte at the starting address, known as big-endian byte order. We show these two formats in Figure 3.9.

Figure 3.9. Little-endian byte order and big-endian byte order for a 16-bit integer.



In this figure, we show increasing memory addresses going from right to left in the top, and from left to right in the bottom. We also show the most significant bit (MSB) as the leftmost bit of the 16-bit value and the least significant bit (LSB) as the rightmost bit.

The terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.

Unfortunately, there is no standard between these two byte orderings and we encounter systems that use both formats. We refer to the byte ordering used by a given system as the host byte order. The program shown in Figure 3.10 prints the host byte order.

3.5 Byte Manipulation Functions

There are two groups of functions that operate on multibyte fields, without interpreting the data, and without a string. We need these types of functions when dealing with socket address structures because we need to manip contain bytes of 0, but are not C character strings. The functions beginning with str (for string), defined by incl terminated C character strings.

The first group of functions, whose names begin with b (for byte), are from 4.2BSD and are still provided by a functions. The second group of functions, whose names begin with mem (for memory), are from the ANSI C s supports an ANSI C library.

We first show the Berkeley-derived functions, although the only one we use in this text is bzero. (We use it be remember than the three-argument memset function, as explained on p. 8.) You may encounter the other two fu applications.

```
#include <strings.h>
void bzero(void *dest, size_t nbytes);
void bcopy(const void *src, void *dest, size_t nbytes);
```

```
int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
```

Returns: 0 if equal, nonzero if unequal

This is our first encounter with the ANSI C const qualifier. In the three uses here, it indicates that what is point
ptr1, and ptr2, is not modified by the function. Worded another way, the memory pointed to by the const point

bzero sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket
number of bytes from the source to the destination. bcmp compares two arbitrary byte strings. The return value
otherwise, it is nonzero.

3.6 inet_aton, inet_addr, and inet_ntoa Functions

We will describe two groups of address conversion functions in this section and the next. They convert Interne
prefer to use) and network byte ordered binary values (values that are stored in socket address structures).

- inet_aton, inet_ntoa, and inet_addr convert an IPv4 address from a dotted-decimal string (e.g., "206.168.112
  value. You will probably encounter these functions in lots of existing code.
- The newer functions, inet_pton and inet_ntop, handle both IPv4 and IPv6 addresses. We describe these two
  throughout the text.

```
#include <arpa/inet.h>
```

```
int inet_aton(const char *strptr, struct in_addr *addrptr);
```

Returns: 1 if string was valid, 0 on error

```
in_addr_t inet_addr(const char *strptr);
```

Returns: 32-bit binary network byte ordered IPv4 address; INADDR_NONE if error

```
char *inet_ntoa(struct in_addr inaddr);
```

Returns: pointer to dotted-decimal string

The first of these, inet_aton, converts the C character string pointed to by

strptr into its 32-bit binary network byte ordered value, which is stored through the pointer addrptr. If successful, 1 is returned; otherwise, 0 is returned.

An undocumented feature of inet_aton is that if addrptr is a null pointer, the function still performs its validation of the input string but does not store any result.

inet_addr does the same conversion, returning the 32-bit binary network byte ordered value

as the return value. The problem with this function is that all 2^{32} possible binary values are valid IP addresses (0.0.0.0 through 255.255.255.255), but the function returns the constant INADDR_NONE (typically 32 one-bits) on an error. This means the dotted-decimal string 255.255.255.255 (the IPv4 limited broadcast address, Section 20.2) cannot be handled by this function since its binary value appears to indicate failure of the function.

A potential problem with inet_addr is that some man pages state that it returns 1 on an error, instead of INADDR_NONE. This can lead to problems, depending on the C compiler, when comparing the return value of the function (an unsigned value) to a negative constant.

Today, inet_addr is deprecated and any new code should use inet_aton instead. Better still is to use the newer functions described in the next section, which handle both IPv4 and IPv6.

The inet_ntoa function converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string.

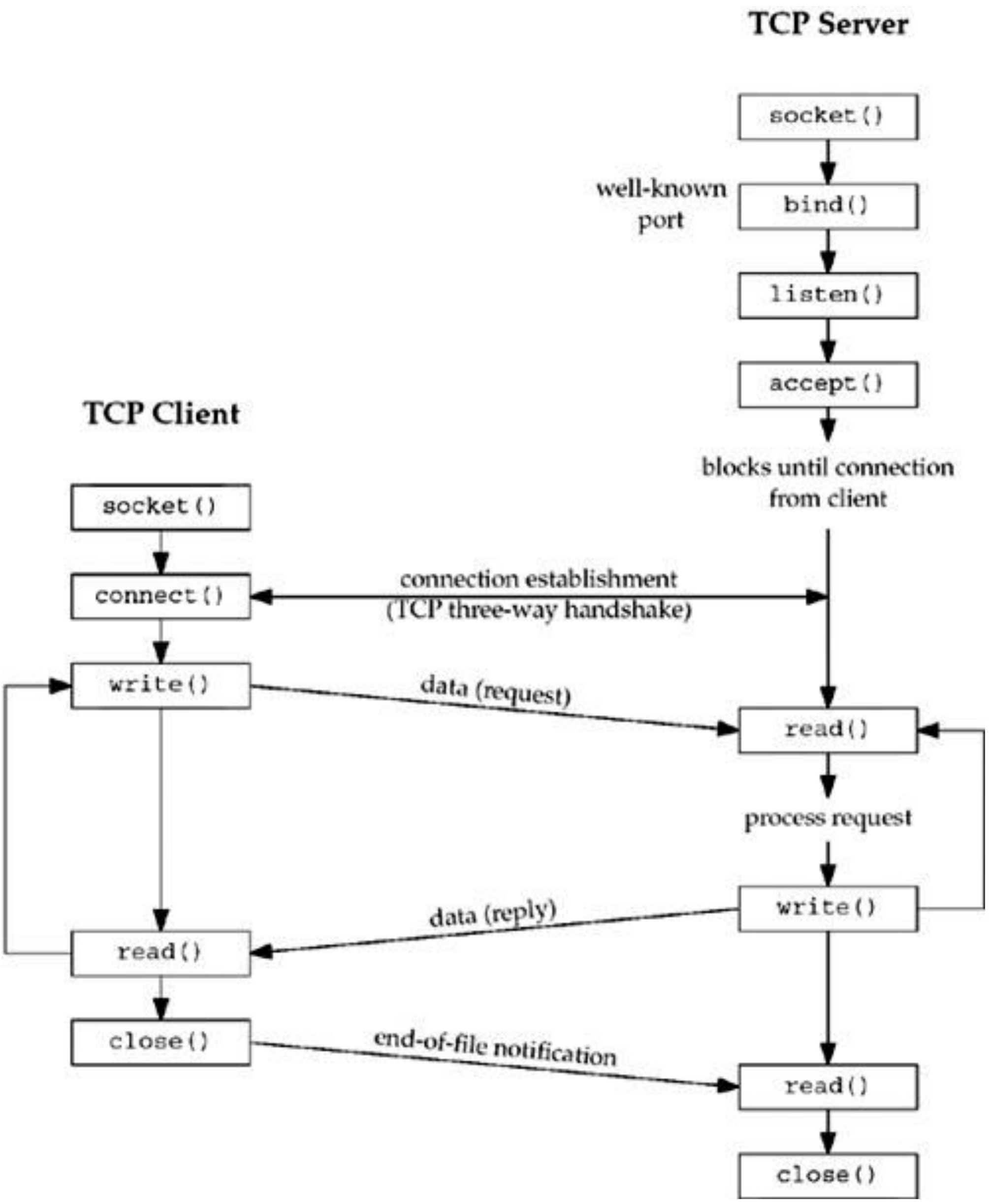
4.1 Introduction

This chapter describes the elementary socket functions required to write a complete TCP client and server. We will first describe all the elementary socket functions that we will be using and then develop the client and server in the next chapter. We will work with this client and server throughout the text, enhancing it many times (Figures 1.12 and 1.13).

We will also describe concurrent servers, a common Unix technique for providing concurrency when numerous clients are connected to the same server at the same time. Each client connection causes the server to fork a new process just for that client. In this chapter, we consider only the one-process-per-client model using fork, but we will consider a different one-thread-per-client model when we describe threads in Chapter 26.

Figure 4.1 shows a timeline of the typical scenario that takes place between a TCP client and server. First, the server is started, then sometime later, a client is started that connects to the server. We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. The server then closes its end of the connection and either terminates or waits for a new client connection.

Figure 4.1. Socket functions for elementary TCP client/server.



4.2 socket Function

To perform network I/O, the first thing a process must do is call the socket function, specifying the type of con
UDP using IPv6, Unix domain stream protocol, etc.).

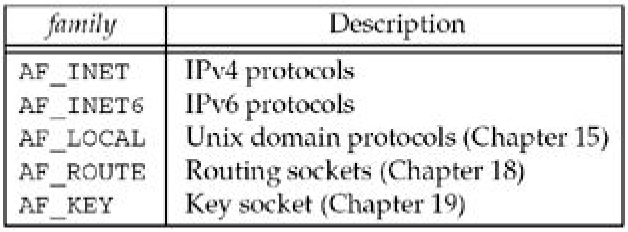
```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
```

Returns: non-negative descriptor if OK, -1 on error

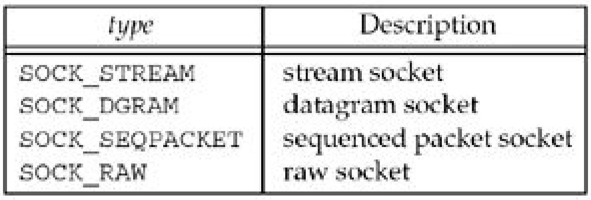
family specifies the protocol family and is one of the constants shown in Figure 4.2. This argument is often ref
type is one of the constants shown in Figure 4.3. The protocol argument to the socket function should be set to
0 to select the system's default for the given combination of family and type.

Figure 4.2. Protocol family constants for socket function.

Figure 4.3. type of socket for socket function.



family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)



type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

Figure 4.2

Figure 4.3

4.3 connect Function The connect function is used by a TCP client to establish a connection with a TCP serve

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Returns: 0 if OK, -1 on error

sockfd is a socket descriptor returned by the socket function. The second and third arguments are a pointer to a described in Section 3.3. The socket address structure must contain the IP address and port number of the serve 1.5.

The client does not have to call bind (which we will describe in the next section) before calling connect: the ke source IP address if necessary.

In the case of a TCP socket, the connect function initiates TCP's three-way handshake (Section 2.6). The func established or an error occurs. There are several different error returns possible.

- If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned. 4.4BSD, for example, 6 seconds later, and another 24 seconds later (p. 828 of TCPv2). If no response is received after a total systems provide administrative control over this timeout; see Appendix E of TCPv1.

- If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for conn (i.e., the server process is probably not running). This is a hard error and the error ECONNREFUSED i received. An RST is a type of TCP segment that is sent by TCP when something is wrong. Three conc arrives for a port that has no listening server (what we just described), when TCP wants to abort an exis segment for a connection that does not exist. (TCPv1 [pp. 246 250] contains additional information.)

- If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is conside message but keeps sending SYNs with the same time between each SYN as in the first scenario. If no r time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either EHOSTUNRE the remote system is not reachable by any route in the local system's forwarding table, or that the conne

4.4 bind Function

The bind function assigns a local protocol address to a socket. With the Internet protocols, the protocol address
number.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

Returns: 0 if OK,-1 on error

4.5 listen Function

The listen function is called only by a TCP server and it performs two actions:

. When a socket is created by the socket function, it is assumed to be an active socket, that is, a client socket t
converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming
terms of the TCP state transition diagram (Figure 2.4), the call to listen moves the socket from the CLO

. The second argument to this function specifies the maximum number of connections the kernel should queu

```
#include <sys/socket.h>
#int listen (int sockfd, int backlog);
```

Returns: 0 if OK, -1 on error

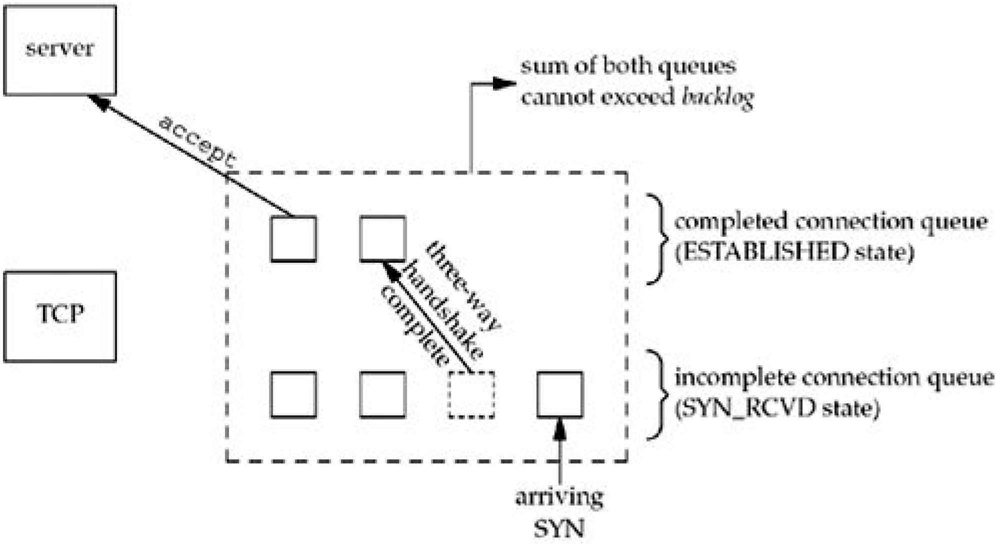
This function is normally called after both the socket and bind functions and must be called before calling the a

To understand the backlog argument, we must realize that for a given listening socket, the kernel maintains two

. An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for wh three-way handshake. These sockets are in the SYN_RCVD state (Figure 2.4).

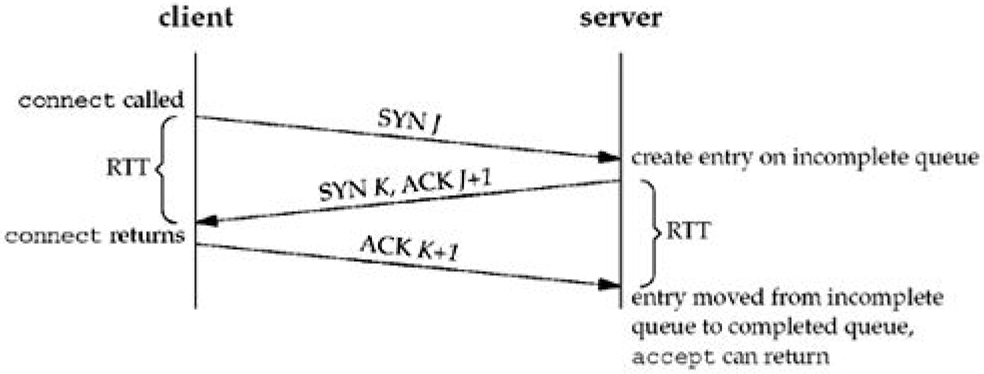
. A completed connection queue, which contains an entry for each client with whom the TCP three-way hands ESTABLISHED state (Figure 2.4).

Figure 4.7 depicts these two queues for a given listening socket. Figure 4.7. The two queues maintained by TC



When an entry is created on the incomplete queue, the parameters from the listen socket are copied over to the newly created connection. The connection creation mechanism is completely automatic; the server process is not involved. Figure 4.8 depicts the packets exchanged during the connection establishment with these two queues.

Figure 4.8. TCP three-way handshake and the two queues for a listening socket.



When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three-way handshake: the server's SYN with an ACK of the client's SYN. This entry will remain on the incomplete queue until the third segment of the three-way handshake arrives (the client's ACK of the server's SYN), or until the entry times out. (Berkeley-derived implementations have a timeout of 75 seconds for these incomplete entries.) If the three-way handshake completes normally, the entry moves from the incomplete queue to the end of the completed queue. When the process calls accept, which we will describe in the next section, the first entry on the completed queue is returned to the process, or if the queue is empty, the process is put to sleep until an entry is placed onto the completed queue.

4.6 accept Function

accept is called by a TCP server to return the next completed connection from the front of the completed conne
connection queue is empty, the process is put to sleep (assuming the default of a blocking socket).

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Returns: non-negative descriptor if OK, -1 on error

The cliaddr and addrlen arguments are used to return the protocol address of the connected peer process (the cl
(Section 3.3): Before the call, we set the integer value referenced by *addrlen to the size of the socket address s
integer value contains the actual number of bytes stored by the kernel in the socket address structure.

If accept is successful, its return value is a brand-new descriptor automatically created by the kernel. This new
the client. When discussing accept, we call the first argument to accept the listening socket (the descriptor crea
argument to both bind and listen), and we call the return value from accept the connected socket. It is importan
given server normally creates only one listening socket, which then exists for the lifetime of the server. The ke
client connection that is accepted (i.e., for which the TCP three-way handshake completes). When the server is
socket is closed.

4.8 Concurrent Servers

The server in Figure 4.11 is an iterative server. For something as simple as a daytime server, this is fine. But when a client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time. The simplest way to write a concurrent server under Unix is to fork a child process to handle each client. Figure 4.13 shows the outline for a typical concurrent server.

Figure 4.13 Outline for typical concurrent server.

```
pid_t pid;
int   listenfd,  connfd;
listenfd = Socket( ... );
   /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
```

```
Listen(listenfd, LISTENQ);
for ( ; ; ) {
    connfd = Accept (listenfd, ... );    /* probably blocks */
    if( (pid = Fork()) == 0) {

}

   Close(listenfd);
   doit(connfd);
   Close(connfd);
   exit(0);
}
Close(connfd);
/* child closes listening socket */
/* process the request */
/* done with this client */
/* child terminates */
/* parent closes connected socket */
```

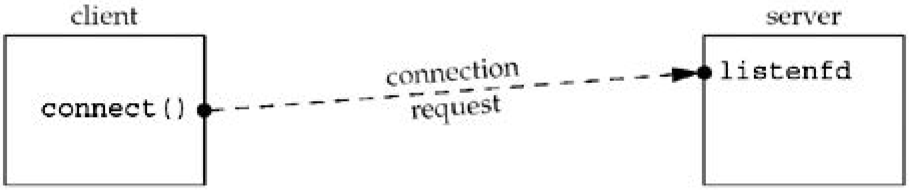
When a connection is established, accept returns, the server calls fork, and the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket). The parent closes the connected socket since the child handles the new client.

In Figure 4.13, we assume that the function doit does whatever is required to service the client. When this function returns, we explicitly close the connected socket in the child. This is not required since the next statement calls exit, and part of process termination is to close all open descriptors by the kernel. Whether to include this explicit call to close or not is a matter of personal programming taste.

We said in Section 2.6 that calling close on a TCP socket causes a FIN to be sent, followed by the normal TCP connection termination sequence. Why doesn't the close of connfd in Figure 4.13 by the parent terminate its connection with the client? To understand what's happening, we must understand that every file or socket has a reference count. The reference count is maintained in the file table entry (pp. 57 60 of APUE). This is a count of the number of descriptors that are currently open that refer to this file or socket. In Figure 4.13, after socket returns, the file table entry associated with listenfd has a reference count of 1. After accept returns, the file table entry associated with connfd has a reference count of 1. But, after fork returns, both descriptors are shared (i.e., duplicated) between the parent and child, so the file table entries associated with both sockets now have a reference count of 2. Therefore, when the parent closes connfd, it just decrements the reference count from 2 to 1 and that is all. The actual cleanup and de-allocation of the socket does not happen until the reference count reaches 0. This will occur at some time later when the child closes connfd.

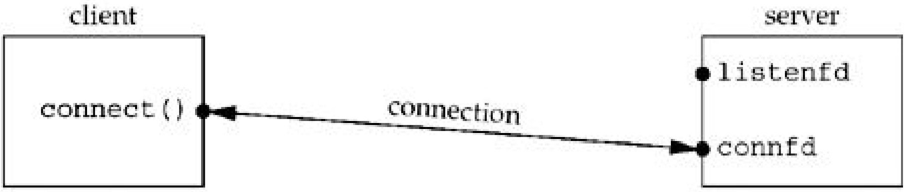
We can also visualize the sockets and connection that occur in Figure 4.13 as follows. First, Figure 4.14 shows the status of the client and server while the server is blocked in the call to accept and the connection request arrives from the client.

Figure 4.14. Status of client/server before call to accept returns.



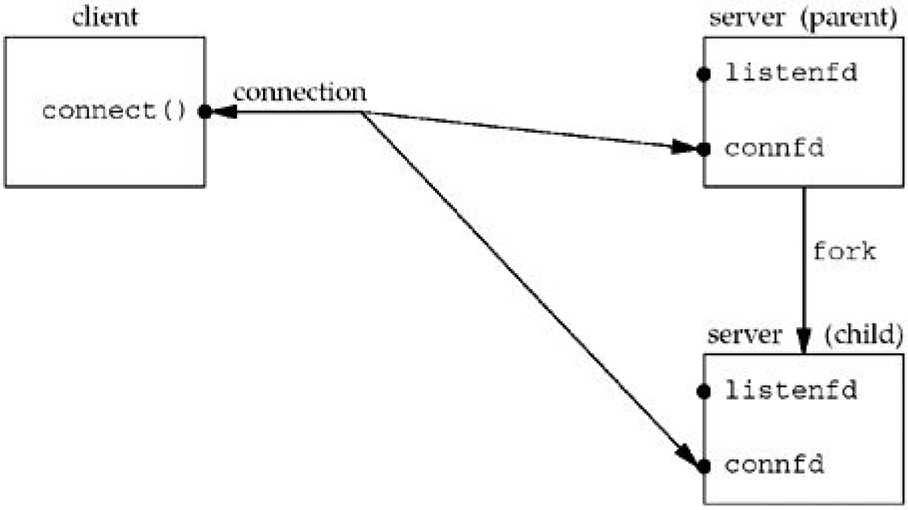
Immediately after accept returns, we have the scenario shown in Figure 4.15. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.

Figure 4.15. Status of client/server after return from accept.



The next step in the concurrent server is to call fork. Figure 4.16 shows the status after fork returns.

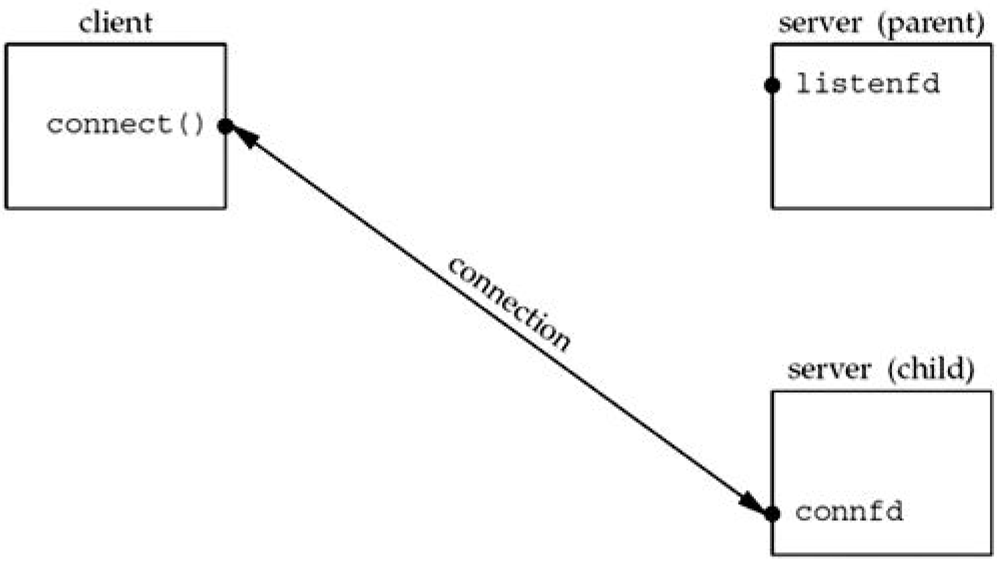
Figure 4.16. Status of client/server after fork returns.



Notice that both descriptors, listenfd and connfd, are shared (duplicated) between the parent and child.

The next step is for the parent to close the connected socket and the child to close the listening socket. This is shown in Figure 4.17.

Figure 4.17. Status of client/server after parent and child close appropriate sockets.



This is the desired final state of the sockets. The child is handling the connection with the client and the parent can call accept again on the listening socket, to handle the next client connection.

4.9 close Function

The normal Unix close function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>
int close (int sockfd);
```

Returns: 0 if OK, -1 on error

4.10 recvfrom and sendto Function

These two functions are similar to the standard read and write functions, but three additional arguments are req

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr * from, socklen_t *addrlen);
ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct sockaddr *to, socklen_t addrler
```

Both return: number of bytes read or written if OK, 1 on error

The first three arguments, sockfd, buff, and nbytes, are identical to the first three arguments for read and write;

from, and number of bytes to read or write.

We will describe the flags argument in Chapter 14 when we discuss the recv, send, recvmsg, and sendmsg func
UDP client/server example in this chapter. For now, we will always set the flags to 0.

The to argument for sendto is a socket address structure containing the protocol address (e.g., IP address and p
size of this socket address structure is specified by addrlen. The recvfrom function fills in the socket address st
address of who sent the datagram. The number of bytes stored in this socket address structure is also returned t
Note that the final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to a

The final two arguments to recvfrom are similar to the final two arguments to accept: The contents of the sock
the datagram (in the case of UDP) or who initiated the connection (in the case of TCP). The final two argumen
to connect: We fill in the socket address structure with the protocol address of where to send the datagram (in t
connection (in the case of TCP).

Both functions return the length of the data that was read or written as the value of the function. In the typical u
return value is the amount of user data in the datagram received.

Writing a datagram of length 0 is acceptable. In the case of UDP, this results in an IP datagram containing an I
bytes for IPv6), an 8-byte UDP header, and no data. This also means that a return value of 0 from recvfrom is a
mean that the peer has closed the connection, as does a return value of 0 from read on a TCP socket. Since UD
closing a UDP connection.

Advanced I/O

6.1 Introduction

In Section 5.12, we saw our TCP client handling two inputs at the same time: standard input and a TCP socket. We encountered a problem when the client was blocked in a call to fgets (on standard input) and the server process was killed. The server TCP correctly sent a FIN to the client TCP, but since the client process was blocked reading from standard input, it never saw the EOF until it read from the socket (possibly much later). What we need is the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e., input is ready to be read, or the descriptor is capable of taking more output). This capability is called I/O multiplexing and is provided by the select and poll functions. We will also cover a newer POSIX variation of the former, called pselect.

Some systems provide more advanced ways for processes to wait for a list of events. A poll device is one mechanism provided in different forms by different vendors. This mechanism will be described in Chapter 14.

I/O multiplexing is typically used in networking applications in the following scenarios:

. □□□When a client is handling multiple descriptors (normally interactive input and a network socket), I/O multiplexing should be used. This is the scenario we described previously.

. □□□It is possible, but rare, for a client to handle multiple sockets at the same time. We will show an example of this using select in Section 16.5 in the context of a Web client.

. □□□If a TCP server handles both a listening socket and its connected sockets, I/O multiplexing is normally used, as we will show in Section 6.8.

. □□□If a server handles both TCP and UDP, I/O multiplexing is normally used. We will show an example of this in Section 8.15.

. □□□If a server handles multiple services and perhaps multiple protocols (e.g., the inetd daemon that we will describe in Section 13.5), I/O multiplexing is normally used.   I/O multiplexing is not limited to network programming. Many nontrivial applications find a need for these techniques.

6.2 I/O Models

Before describing select and poll, we need to step back and look at the bigger picture, examining the basic differences in the five I/O models that are available to us under Unix:

. □□□blocking I/O

. □□□nonblocking I/O

. □□□I/O multiplexing (select and poll)

. □□□signal driven I/O (SIGIO)

. □□□asynchronous I/O (the POSIX aio_functions) You may want to skim this section on your first reading and then refer back to it as you encounter the different I/O models described in more detail in later chapters. As we show in all the examples in this section, there are normally two distinct phases for an input operation:

. Waiting for the data to be ready

. Copying the data from the kernel to the process

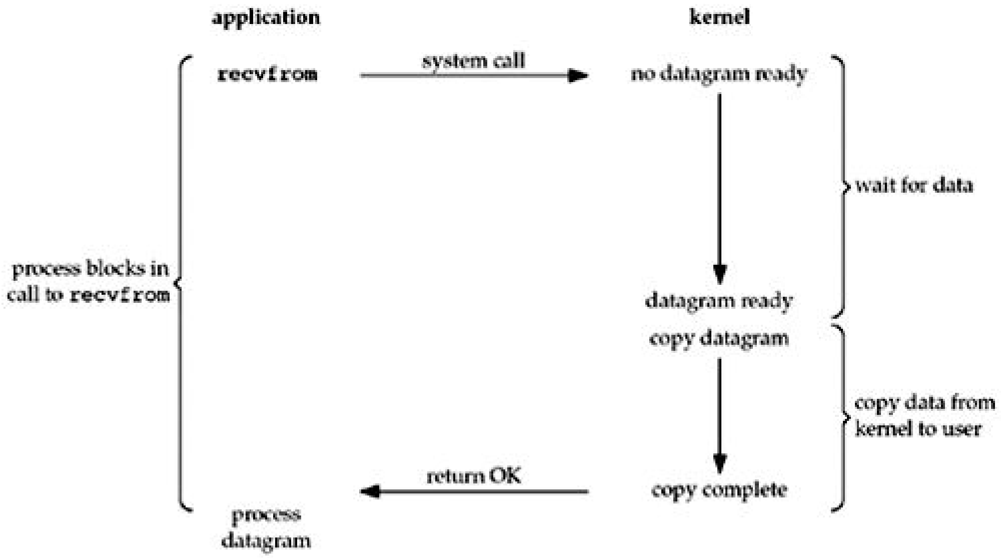
For an input operation on a socket, the first step normally involves waiting for data to arrive on the network. When the packet arrives, it is copied into a buffer within the kernel. The second step is copying this data from the kernel's buffer into our application buffer.

Blocking I/O Model

The most prevalent model for I/O is the blocking I/O model, which we have used for all our examples so far in the text. By default, all sockets are blocking. Using a datagram socket for our examples, we have the scenario shown in Figure 6.1.

Figure 6.1. Blocking I/O model.

We use UDP for this example instead of TCP because with UDP, the concept of data being "ready" to read is simple: either an entire datagram has been received or it has not. With TCP



Nonblocking I/O Model

When we set a socket to be nonblocking, we are telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead." We will describe nonblocking I/O in Chapter 16, but Figure 6.2 shows a summary of the example we are considering.

Figure 6.2. Nonblocking I/O model.

The first three times that we call recvfrom, there is no data to return, so the kernel immediately returns an error of EWOULDBLOCK instead. The fourth time we call recvfrom, a datagram is ready, it is copied into our application buffer, and recvfrom returns successfully. We then process the data.

When an application sits in a loop calling recvfrom on a nonblocking descriptor like this, it is called polling. The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

I/O Multiplexing Model

With I/O multiplexing, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call. Figure 6.3 is a summary of the I/O multiplexing model.

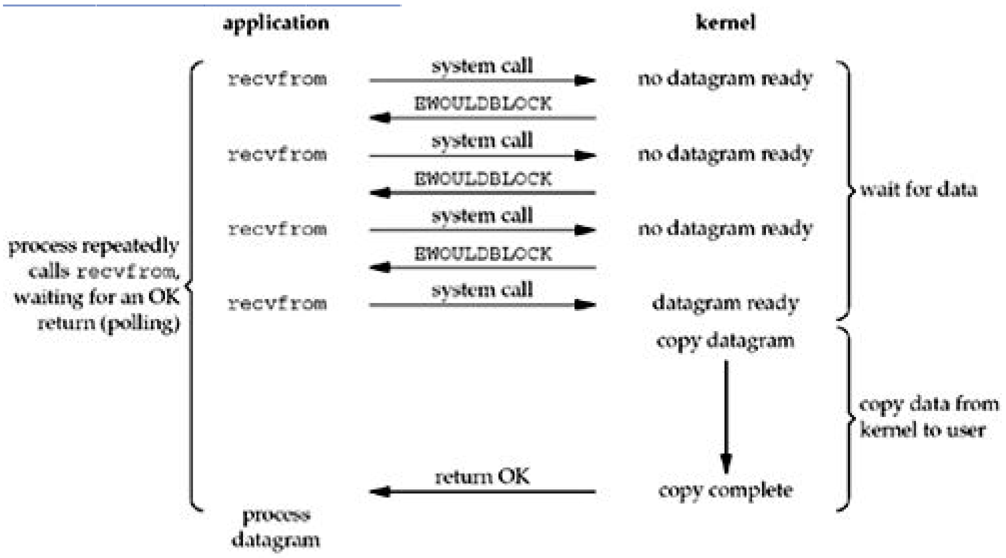
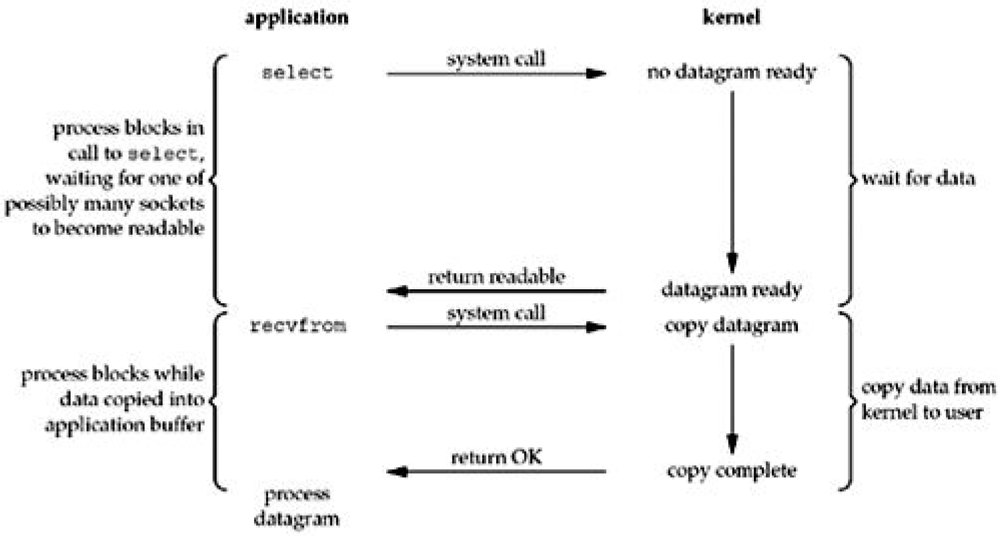


Figure 6.3. I/O multiplexing model.


application
kernel
select
system call
no datagram ready
process blocks in call to select, waiting for one of possibly many sockets to become readable
wait for data
return readable
datagram ready
recvfrom
system call
copy datagram
process blocks while data copied into application buffer
copy data from kernel to user
return OK
copy complete
process datagram

We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.

Comparing Figure 6.3 to Figure 6.1, there does not appear to be any advantage, and in fact, there is a slight disadvantage because using select requires two system calls instead of one. But the advantage in using select, which we will see later in this chapter, is that we can wait for more than one descriptor to be ready.

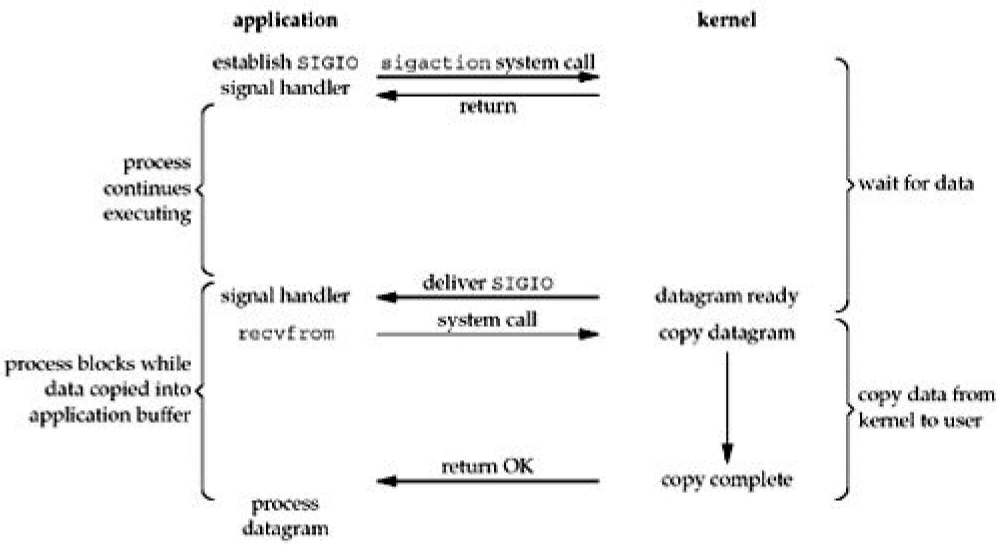
Another closely related I/O model is to use multithreading with blocking I/O. That model very closely resembles the model described above, except that instead of using select to block on multiple file descriptors, the program uses multiple threads (one per file descriptor), and each thread is then free to call blocking system calls like recvfrom.

Signal-Driven I/O Model

We can also use signals, telling the kernel to notify us with the SIGIO signal when the descriptor is ready. We call this signal-driven I/O and show a summary of it in Figure 6.4.

Figure 6.4. Signal-Driven I/O model.

We first enable the socket for signal-driven I/O (as we will describe in Section 25.2) and install a signal handler using the sigaction system call. The return from this system call is



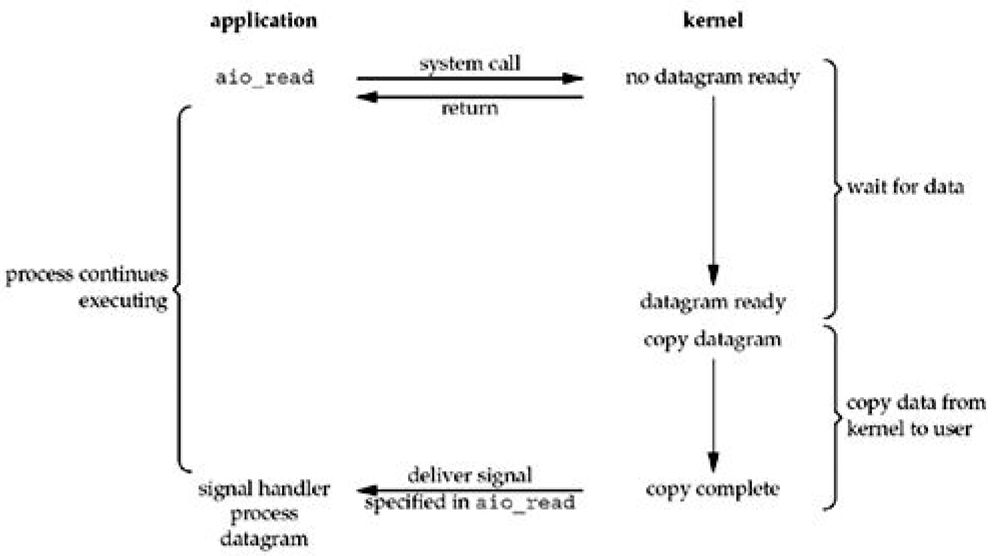
immediate and our process continues; it is not blocked. When the datagram is ready to be read, the SIGIO signal is generated for our process. We can either read the datagram from the signal handler by calling recvfrom and then notify the main loop that the data is ready to be processed (this is what we will do in Section 25.3), or we can notify the main loop and let it read the datagram.

Regardless of how we handle the signal, the advantage to this model is that we are not blocked while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

Asynchronous I/O Model

Asynchronous I/O is defined by the POSIX specification, and various differences in the real-time functions that appeared in the various standards which came together to form the current POSIX specification have been reconciled. In general, these functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete. The main difference between this model and the signal-driven I/O model in the previous section is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/O operation is complete. We show an example in Figure 6.5.

Figure 6.5. Asynchronous I/O model.


application
kernel
aio_read
system call
return
no datagram ready
wait for data
process continues executing
datagram ready
copy datagram
copy data from kernel to user
signal handler process datagram
deliver signal specified in aio_read
copy complete

We call aio_read (the POSIX asynchronous I/O functions begin with aio_ or lio_) and pass the kernel the descriptor, buffer pointer, buffer size (the same three arguments for read), file offset (similar to lseek), and how to notify us when the entire operation is complete. This system call returns immediately and our process is not blocked while waiting for the I/O to complete. We assume in this example that we ask the kernel to generate some signal when the operation is complete. This signal is not generated until the data has been copied into our application buffer, which is different from the signal-driven I/O model.

As of this writing, few systems support POSIX asynchronous I/O. We are not certain, for example, if systems will support it for sockets. Our use of it here is as an example to compare against the signal-driven I/O model.

6.3 select Function

This function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wa events occurs or when a specified amount of time has passed.

As an example, we can call select and tell the kernel to return only when:

. □□□Any of the descriptors in the set {1, 4, 5} are ready for reading

. □□□Any of the descriptors in the set {2, 7} are ready for writing

. □□□Any of the descriptors in the set {1, 4} have an exception condition pending

. □□□10.2 seconds have elapsed That is, we tell the kernel what descriptors we are interested i and how long to wait. The descriptors in which we are interested are not restricted to sockets; any descr derived implementations have always allowed I/O multiplexing with any descriptor. SVR3 originally li STREAMS devices (Chapter 31), but this limitation was removed with SVR4.

```
#include <sys/select.h>
```

```
#include <sys/time.h>
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);
```

Returns: positive count of ready descriptors, 0 on timeout, 1 on error

We start our description of this function with its final argument, which tells the kernel how long to wait for one timeval structure specifies the number of seconds and microseconds.

```
struct timeval  {
  long   tv_sec;         /* seconds */
  long   tv_usec;        /* microseconds */
};
```

There are three possibilities:

. Wait forever Return only when one of the specified descriptors is ready for I/O. For this, we specify the time

. Wait up to a fixed amount of time Return when one of the specified descriptors is ready for I/O, but do not w microseconds specified in the timeval structure pointed to by the timeout argument.

3. Do not wait at all Return immediately after checking the descriptors. This is called polling. To specify this, the timeout argument must point to a timeval structure and the timer value (the number of seconds and microseconds specified by the structure) must be 0.

A design problem is how to specify one or more descriptor values for each of these three arguments. select uses descriptor sets, typically an array of integers, with each bit in each integer corresponding to a descriptor. For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63, and so on. All the implementation details are irrelevant to the application and are hidden in the fd_set datatype and the following four macros:

```
void FD_ZERO(fd_set *fdset); /* clear all bits in fdset*/

void FD_SET(int fd, fd_set *fdset); /* turn on the bit for fd in fdset */

void FD_ZERO(fd_set *fdset); /* clear all bits in fdset */

void FD_CLR(int fd, fd_set *fdset); /* turn off the bit for fd in fdset */

int FD_ISSET(int fd, fd_set *fdset); /* is the bit for fd on in fdset ? */
```

6.10 poll Function

The poll function originated with SVR3 and was originally limited to STREAMS devices (Chapter 31). SVR4 with any descriptor. poll provides functionality that is similar to select, but poll provides additional informatio

```
#include <poll.h>
int poll (struct pollfd *fdarray, unsigned long nfds, int timeout);
```

Returns: count of ready descriptors, 0 on timeout, 1 on error

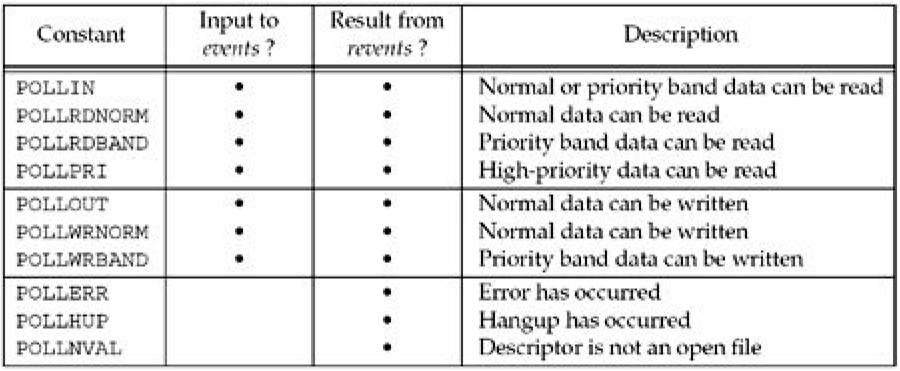
The first argument is a pointer to the first element of an array of structures. Each element of the array is a pollf tested for a given descriptor, fd.

```
struct pollfd {
  int     fd;       /* descriptor to check */
  short   events;   /* events of interest on fd */
  short   revents;  /* events that occurred on fd */
};
```

The conditions to be tested are specified by the events member, and the function returns the status for that desc (Having two variables per descriptor, one a value and one a result, avoids value-result arguments. Recall that t result.) Each of these two members is composed of one or more bits that specify a certain condition. Figure 6.2 flag and to test the revents flag against.

Figure 6.23. Input events and returned revents for poll.

We have divided this figure into three sections: The first four constants deal with input, the next three deal wit Notice that the final three cannot be set in events, but are always returned in revents when the corresponding c



Constant	Input to *events* ?	Result from *revents* ?	Description
POLLIN	•	•	Normal or priority band data can be read
POLLRDNORM	•	•	Normal data can be read
POLLRDBAND	•	•	Priority band data can be read
POLLPRI	•	•	High-priority data can be read
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	Error has occurred
POLLHUP		•	Hangup has occurred
POLLNVAL		•	Descriptor is not an open file

6.11 pselect function

#include <sys/select.h>

int pselect(int *nfds*, fd_set **readfds*, fd_set **writefds*,

fd_set ***exceptfds*, const struct timespec ***timeout***,**

const sigset_t ***sigmask*);**

DESCRIPTION

select() and **pselect**() allow a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform the corresponding I/O operation (e.g., **read**(2)) without blocking.

The operation of **select**() and **pselect**() is identical, with three differences:

Tag	Description
(i)	**select**() uses a timeout that is a *struct timeval* (with seconds and microseconds), while **pselect**() uses a *struct timespec* (with seconds and nanoseconds).
(ii)	**select**() may update the *timeout* argument to indicate how much time was left.**pselect**() does not change this argument.
(iii)	**select**() has no *sigmask* argument, and behaves as **pselect**() called with NULL*sigmask*.

Three independent sets of file descriptors are watched. Those listed in *readfds* will be watched to see if characters become available for reading (more precisely, to see if a read will not block; in particular, a file descriptor is also ready on end-of-file), those in *writefds* will be watched to see if a write will not block, and those in *exceptfds* will be watched for exceptions. On exit, the sets are modified in place to indicate which file descriptors actually changed status. Each of the three file descriptor sets may be specified as NULL if no file descriptors are to be watched for the corresponding class of events.

Four macros are provided to manipulate the sets. **FD_ZERO**() clears a set. **FD_SET**() and **FD_CLR**() respectively add and remove a given file descriptor from a set. **FD_ISSET**() tests to see if a file descriptor is part of the set; this is useful after **select**() returns.

nfds is the highest-numbered file descriptor in any of the three sets, plus 1.

timeout is an upper bound on the amount of time elapsed before **select**() returns. It may be zero, causing **select**() to return immediately. (This is useful for polling.) If *timeout* is NULL (no timeout), **select**() can block indefinitely.

sigmask is a pointer to a signal mask (see **sigprocmask**(2)); if it is not NULL, then **pselect**() first replaces the current signal mask by the one pointed to by *sigmask*, then does the ‘select’ function, and then restores the original signal mask.

readv & writev

NAME

readv, writev - read or write data into multiple buffers

SYNOPSIS

#include <sys/uio.h>

ssize_t readv(int *fd*, **const struct iovec ****vector*, **int** *count***);**

ssize_t writev(int *fd*, **const struct iovec ****vector*, **int** *count***);**

DESCRIPTION

The**readv**() function reads *count* blocks from the file associated with the file descriptor *fd* into the multiple buffers described by *vector*.

The **writev**() function writes at most *count* blocks described by *vector* to the file associated with the file descriptor*fd*.

The pointer *vector* points to a *struct iovec* defined in *<sys/uio.h>* as :

```
struct iovec {
   void *iov_base;   /* Starting address */
   size_t iov_len;   /* Number of bytes */
};
```

Buffers are processed in the order specified. The **readv**() function works just like **read**(2) except that multiple buffers are filled.

The **writev**() function works just like **write**(2) except that multiple buffers are written out.

RETURN VALUE

On success, the **readv**() function returns the number of bytes read; the **writev**() function returns the number of bytes written. On error, -1 is returned, and *errno* is set appropriately.

ERRORS

The errors are as given for **read**(2) and **write**(2). Additionally the following error is defined:

Tag	Description
EINVAL	The sum of the *iov_len* values overflows an *ssize_t* value. Or, the vector count *count*is less than zero or greater than the permitted maximum.

14.9. Memory-Mapped I/O

Memory-mapped I/O lets us map a file on disk into a buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file. This lets us perform I/O without using read or write.

Memory-mapped I/O has been in use with virtual memory systems for many years. In 1981, 4.1BSD provided a different form of memory-mapped I/O with its vread and vwrite functions. These two functions were then removed in 4.2BSD and were intended to be replaced with the mmap function. The mmap function, however, was not included with 4.2BSD (for reasons described in Section 2.5 of McKusick et al. [1996]). Gingell, Moran, and Shannon [1987] describe one implementation of mmap. The mmap function is included in the memory-mapped files option in the Single UNIX Specification and is required on all XSI-conforming systems; most UNIX systems support it.

To use this feature, we have to tell the kernel to map a given file to a region in memory. This is done by the mmap function.

[View full width]

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flag, int filedes,
        off_t off );
```

Returns: starting address of mapped region if OK, MAP_FAILED on error

The *addr* argument lets us specify the address of where we want the mapped region to start. We normally set this to 0 to allow the system to choose the starting address. The return value of this function is the starting address of the mapped area.

The *filedes* argument is the file descriptor specifying the file that is to be mapped. We have to open this file before we can map it into the address space. The *len* argument is the number of bytes to map, and *off* is the starting offset in the file of the bytes to map. (Some restrictions on the value of *off* are described later.)

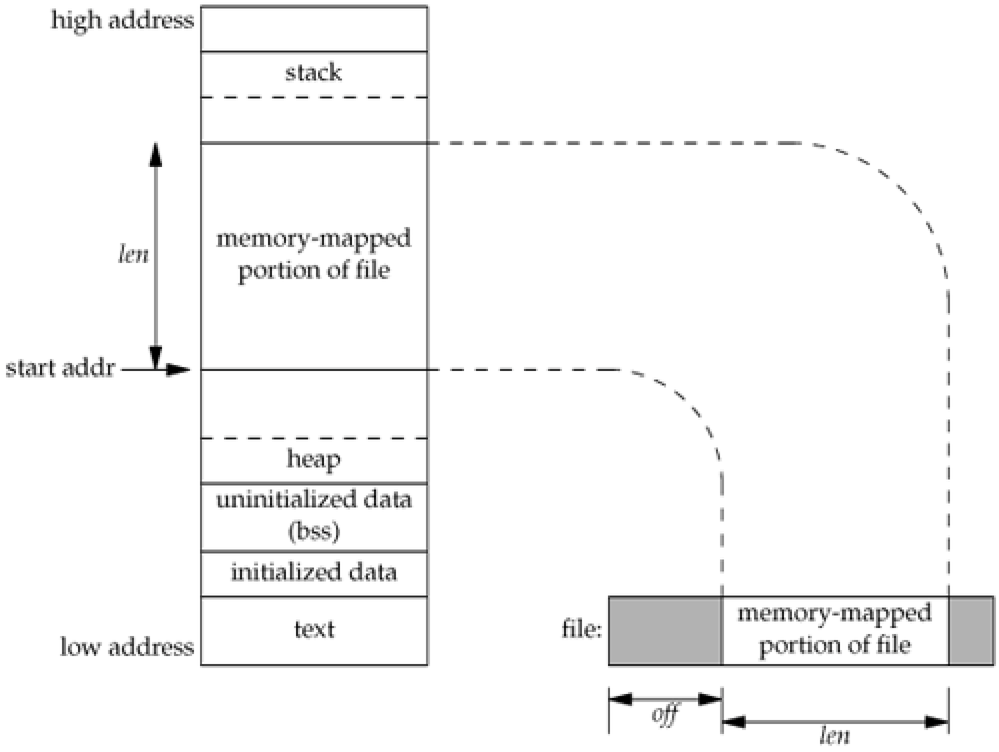
The *prot* argument specifies the protection of the mapped region.

We can specify the protection as either PROT_NONE or the bitwise OR of any combination of PROT_READ, PROT_WRITE, and PROT_EXEC. The protection specified for a region can't allow more access than the open mode of the file. For example, we can't specify PROT_WRITE if the file was opened read-only.

Before looking at the *flag* argument, let's see what's going on here. Figure 14.31 shows a memory-mapped file. (Recall the memory layout of a typical process, Figure 7.6.) In this figure, "start addr" is the return value from mmap. We have shown the mapped memory being somewhere between the heap and the stack: this is an implementation detail and may differ from one implementation to the next.

Figure 14.31. Example of a memory-mapped file

[View full size image]



The *flag* argument affects various attributes of the mapped region.

MAP_FIXED The return value must equal *addr*. Use of this flag is discouraged, as it hinders portability. If this flag is not specified and if *addr* is nonzero, then the kernel uses *addr* as a hint of where to place the mapped region, but there is no guarantee that the requested address will be used. Maximum portability is obtained by specifying *addr* as 0.

Support for the MAP_FIXED flag is optional on POSIX-conforming systems, but required on XSI-conforming systems.

MAP_SHARED This flag describes the disposition of store operations into the mapped region by this process. This flag specifies that store operations modify the mapped filethat is, a store operation is equivalent to a write to the file. Either this flag or the next (MAP_PRIVATE), but not both, must be specified.

MAP_PRIVATE This flag says that store operations into the mapped region cause a private copy of the mapped file to be created. All successive references to the mapped region then reference the copy. (One use of this flag is for a debugger that maps the text portion of a program file but allows the user to modify the instructions. Any modifications affect the copy, not the original program file.)

Each implementation has additional MAP_xxx flag values, which are specific to that implementation. Check the mmap(2) manual page on your system for details.

The value of *off* and the value of *addr* (if MAP_FIXED is specified) are required to be multiples of the system's virtual memory page size. This value can be obtained from the sysconf function with an argument of _SC_PAGESIZE or _SC_PAGE_SIZE. Since *off* and *addr* are often specified as 0, this requirement is not a big deal.

Since the starting offset of the mapped file is tied to the system's virtual memory page size, what happens if the length of the mapped region isn't a multiple of the page size? Assume that the file size is 12 bytes and that the system's page size is 512 bytes. In this case, the system normally provides a mapped region of 512 bytes, and the final 500 bytes of this region are set to 0. We can modify the final 500 bytes, but any changes we make to them are not reflected in the file.

Two signals are normally used with mapped regions. SIGSEGV is the signal normally used to indicate that we have tried to access memory that is not available to us. This signal can also be generated if we try to store into a mapped region that we specified to mmap as read-only. The SIGBUS signal can be generated if we access a portion of the mapped region that does not make sense at the time of the access. For example, assume that we map a file using the file's size, but before we reference the mapped region, the file's size is truncated by some other process. If we then try to access the memory-mapped region corresponding to the end portion of the file that was truncated, we'll receive SIGBUS.

A memory-mapped region is inherited by a child across a fork (since it's part of the parent's address space), but for the same reason, is not inherited by the new program across an exec.

We can change the permissions on an existing mapping by calling mprotect.

```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

Returns: 0 if OK, 1 on error

The legal values for *prot* are the same as those for mmap (Figure 14.30). The address argument must be an integral multiple of the system's page size.

Figure 14.30. Protection of memory-mapped region

prot	Description
PROT_READ	Region can be read.
PROT_WRITE	Region can be written.
PROT_EXEC	Region can be executed.
PROT_NONE	Region cannot be accessed.

The mprotect function is included as part of the memory protection option in the Single UNIX Specification, but all XSI-conforming systems are required to support it.

If the pages in a shared mapping have been modified, we can call msync to flush the changes to the file that backs the mapping. The msync function is similar to fsync (Section 3.13), but works on memory-mapped regions.

```
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);
```

Returns: 0 if OK, 1 on error

If the mapping is private, the file mapped is not modified. As with the other memory-mapped functions, the address must be aligned on a page boundary.

The *flags* argument allows us some control over how the memory is flushed. We can specify the MS_ASYNC flag to simply schedule the pages to be written. If we want to wait for the writes to complete before returning, we can use the MS_SYNC flag.
Either MS_ASYNC or MS_SYNC must be specified.

An optional flag, MS_INVALIDATE, lets us tell the operating system to discard any pages that are out of sync with the underlying storage. Some implementations will discard all pages in the specified range when we use this flag, but this behavior is not required.

A memory-mapped region is automatically unmapped when the process terminates or by calling munmap directly. Closing the file descriptor *filedes* does not unmap the region.

```
#include <sys/mman.h>

int munmap(caddr_t addr, size_t len);
```

Returns: 0 if OK, 1 on error

munmap does not affect the object that was mappedthat is, the call to munmap does not cause the contents of the mapped region to be written to the disk file. The updating of the disk file for a MAP_SHARED region happens automatically by the kernel's virtual memory algorithm as we store into the memory-mapped region. Modifications to memory in a MAP_PRIVATE region are discarded when the region is unmapped.

File and Record Locking

IRIX supports the ability to place a lock upon an entire file or upon a range of bytes within a file. Programs must cooperate in respecting record locks. A file lock can be made mandatory but only at a cost in performance. For these reasons, file and record locking should normally be seen as a synchronization mechanism, not a security mechanism.

The chapter includes these topics:

- "Overview of File and Record Locking" presents an introduction to locking mechanisms.
- "Controlling File Access With File Permissions" discusses the relationship of file permissions to exclusive file access.
- "Using Record Locking" discusses the use of file and record locks to get exclusive data access.
- "Enforcing Mandatory Locking" describes how file locks can be made mandatory on programs that do not use locking.
- "Record Locking Across Multiple Systems" discusses how file locking can be extended to NFS-mounted files.

Overview of File and Record Locking

Simultaneous access to file data is characteristic of many multiprocess, multithreaded, or real-time applications.The purpose of the file and record locking facility is to provide a way for programs to synchronize their use of common file data.

Advisory file and record locking can be used to coordinate independent, unrelated processes. In mandatory locking, on the other hand, the standard I/O subroutines and I/O system calls enforce the locking protocol. Mandatory locking keeps unrelated programs from accessing data out of sequence, at some cost of access speed.

The system functions used in file and record locking are summarized in Table 7-1.

Table 7-1. Functions for File and Record Locking

Function Name	Purpose and Operation
fcntl(2), fcntl(5)	General function for modifying an open file descriptor; can be used to set file and record locks.
lockf(3C), lockf(3F)	Library function to set and remove file and record locks on open files (SVR4 compatible).
flock(3B)	Library function to set and remove file and record locks on open files (BSD compatible).
chmod(1), chmod(2)	Command and system function that can enable mandatory file locking on a specified file.

Terminology

The discussion of file and record locking depends on the terms defined in this section.

Record

A record is any contiguous sequence of bytes in a file. The UNIX operating system does not impose any record structure on files. The boundaries of records are defined by the programs that use the files. Within a single file, a record as defined by one process can overlap partially or completely on a record as defined by some other process.

Read (Shared) Lock

A read lock keeps a record from changing while one or more processes read the data. If a process holds a read lock, it may assume that no other process can alter that record at the same time. A read lock is also a shared lock because more than one process can place a read lock on the same record or on a record that overlaps a read-locked record. No process, however, can have a write lock that overlaps a read lock.

Write (Exclusive) Lock

A write lock is used to gain complete control over a record. A write lock is an exclusive lock because, when a write lock is in place on a record, no other process may read- or write-lock that record or any data that overlaps it. If a process holds a write lock it can assume that no other process will read or write that record at the same time.

Advisory Locking

An advisory lock is visible only when a program explicitly tries to place a conflicting lock. An advisory lock is not visible to the file I/O system functions such as read() and write(). A process that does not test for an advisory lock can violate the terms of the lock, for example, by writing into a locked record.

Advisory locks are useful when all processes make an appropriate record lock request before performing any I/O operation. When all processes use advisory locking, access to the locked data is controlled by the advisory lock requests. The success of advisory locking depends on the cooperation of all processes in enforcing the locking protocol; it is not enforced by the file I/O subsystem.

Mandatory Locking

Mandatory record locking is enforced by the file I/O system functions, and so is effective on unrelated processes that are not part of a cooperating group. Respect for locked records is enforced by the creat(), open(), read(), and write() system calls. When a record is locked, access to that record by any other process is restricted according to the type of lock on the record. Cooperating processes should still request an appropriate record lock before an I/O operation, but an additional check is made by IRIX before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers security against unplanned file use by unrelated programs, but it imposes additional system overhead on access to the controlled files.

Lock Promotion and Demotion

A read lock can be promoted to write-lock status if no other process is holding a read lock in the same record. If processes with pending write locks are waiting for the same record, the lock promotion succeeds and the other (sleeping) processes wait. Demoting a write lock to a read lock can be done at any time.

Because the lockf() function does not support read locks, lock promotion is not applicable to locks set with that call. >

Controlling File Access With File Permissions

The access permissions for each UNIX file control which users can read, write, or execute the file. These access permissions may be set only by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the access permissions for a file. Note that if the permissions for a directory allow anyone to write in the directory, and the “sticky bit” is not included in the permissions, files within that directory can be removed even by a user who does not have read, write, or execute permission for those files.

If your application warrants the use of record locking, make sure that the permissions on your files and directories are also set properly. A record lock, even a mandatory record lock, protects only the records that are locked, while they are locked. Unlocked parts of the files can be corrupted if proper precautions are not taken.

Only a known set of programs or users should be able to read or write a database. This can be enforced through file permissions as follows:

1. Using the chown facility (see the chown(1) and chown(2) reference pages), set the ownership of the critical directories and files to reflect the authorized group ID.
2. Using the chmod facility (see also the chmod(1) and chmod(2) reference pages), set the file permissions of the critical directories and files so that only members of the authorized group have write access (“775” permissions).
3. Using the chown facility, set the accessing program executable files to be owned by the authorized group.
4. Using the chmod facility, set the set-GID bit for each accessing program executable file and to permit execution by anyone (“2755” permissions).

Users who are not members of the authorized group cannot modify the critical directories and files. However, when an ordinary user executes one of the accessing programs, the program automatically adopts the group ID of its owner. The accessing program can create and modify files in the critical directory, but other programs started by an ordinary user cannot.

Using Record Locking

This section covers the following topics:

- “Opening a File for Record Locking”
- “Setting a File Lock”
- “Setting and Removing Record Locks”
- “Getting Lock Information”
- “Deadlock Handling”

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be used, then the file must be opened with at least read access; likewise for write locks and write access.

Example 7-1 opens a file for both read and write access.

Example 7-1. Opening a File for Locked Use

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
int fd;    /* file descriptor */
char *filename;
main(argc, argv)
int argc;
```

```
char *argv[];
{
   extern void exit(), perror();
   /* get database file name from command line and open the
    * file for read and write access.
    */
   if (argc < 2) {
      (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
      exit(2);
   }
   filename = argv[1];
   fd = open(filename, O_RDWR);
   if (fd < 0) {
      perror(filename);
      exit(2);
   }
}
```

The file is now open to perform both locking and I/O functions. The next step is to set a lock.

Setting a File Lock

Several ways exist to set a lock on a file. These methods depend upon how the lock interacts with the rest of the program. Issues of portability and performance need to be considered. Three methods for setting a lock are given here: using the fcntl() system call; using the */usr/group* standards-compatible lockf() library function; and using the BSD compatible flock() library function.

Locking an entire file is just a special case of record locking—one record is locked, which has the size of the entire file. The file is locked starting at a byte offset of zero and size of the maximum file size. This size is beyond any real end-of-file so that no other lock can be placed on the file.

You have a choice of three functions for this operation: the basic fcntl(), the library function lockf(), and the BSD compatible library function flock(). All three functions can interoperate. That is, a lock placed by one is respected by the other two.

Whole-File Lock With fcntl()

The fcntl() function treats a lock length of 0 as meaning "size of file." The function lockWholeFile() in Example 7-2 attempts a specified number of times to obtain a whole-file lock using fcntl(). When the lock is placed, it returns 0; otherwise it returns the error code for the failure.

Example 7-2. Setting a Whole-File Lock With fcntl()

```
#include <fcntl.h>
#include <errno.h>
#define MAX_TRY 10

int
lockWholeFile(int fd, int tries)
{
    int limit = (tries)?tries:MAX_TRY;
    int try;
    struct flock lck;
    lck.l_type = F_WRLCK;        /* write (exclusive) lock */
    lck.l_whence = 0;           /* 0 offset for l_start */
    lck.l_start = 0L;           /* lock starts at BOF */
    lck.l_len = 0L;             /* extent is entire file */
    for (try = 0; try < limit; ++try)
    {
        if ( 0 == fcntl(fd, F_SETLK, &lck) )
            break; /* mission accomplished */
        if ((errno != EAGAIN) && (errno != EACCES))
            break; /* mission impossible */
        sginap(1); /* let lock holder run */
    }
    return errno;
}
```

The following points should be noted in Example 7-2:

- Because fcntl() supports both read and write locks, the type of the lock (F_WRLCK) is specified in the *l_type*.
- The operation code F_SETLK is used to request that the function return if it cannot place the lock. The code F_SETLKW would request that the function suspend until the lock can be placed.
- The starting location of the record is the sum of two fields, *l_whence* and *l_start*. Both must be set to 0 in order to get the starting point to the beginning of the file.

Whole-File Lock With lockf()

Example 7-3 shows a version of the lockWholeFile() function that uses lockf(). Like fcntl(), lockf() treats a record length of 0 as meaning “to end of file.”

Example 7-3. Setting a Whole-File Lock With lockf()

```
#include <unistd.h> /* for F_TLOCK */
#include <fcntl.h>  /* for O_RDWR */
```

```
#include <errno.h>  /* for EAGAIN */
#define MAX_TRY 10

int
lockWholeFile(int fd, int tries)
{
   int limit = (tries)?tries:MAX_TRY;
   int try;
   lseek(fd,0L,SEEK_SET);  /* set start of lock range */
   for (try = 0; try < limit; ++try)
   {
      if (0 == lockf(fd, F_TLOCK, 0L) )
         break; /* mission accomplished */
      if (errno != EAGAIN)
         break; /* mission impossible */
      sginap(1); /* let lock holder run */
   }
   return errno;
}
```

The following points should be noted about Example 7-3:

- The type of lock is not specified, because lockf() only supports exclusive locks.
- The operation code F_TLOCK specifies that the function should return if the lock cannot be placed. The F_LOCK operation would request that the function suspend until the lock could be placed.
- The start of the record is set implicitly by the current file position. That is why lseek() is called, to ensure the correct file position before lockf() is called.

Whole-File Lock With flock()

Example 7-4 displays a third example of the lockWholeFile subroutine, this one using flock().

Example 7-4. Setting a Whole-File Lock With flock()

```
#define _BSD_COMPAT
#include <sys/file.h> /* includes fcntl.h */
#include <errno.h>  /* for EAGAIN */
#define MAX_TRY 10
int
lockWholeFile(int fd, int tries)
{
   int limit = (tries)?tries:MAX_TRY;
   int try;
```

```
  for (try = 0; try < limit; ++try)
  {
    if ( 0 == flock(fd, LOCK_EX+LOCK_NB) )
      break; /* mission accomplished */
    if (errno != EWOULDBLOCK)
      break; /* mission impossible */
    sginap(1); /* let lock holder run */
  }
  return errno;
}
```

The following points should be noted about Example 7-4:

- The compiler variable _BSD_COMPAT is defined in order to get BSD-compatible definitions from standard header files.
- The only use of flock() is to lock an entire file, so there is no attempt to specify the start or length of a record.
- The LOCK_NB flag requests the function to return if the lock cannot be placed. Without this flag the function suspends until the lock can be placed.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file, except that the record does not encompass the entire file contents. This section examines an example problem of dealing with two records (which may be either in the same file or in different files) that must be updated simultaneously so that other processes get a consistent view of the information they contain. This type of problem occurs, for example, when updating the inter-record pointers in a doubly linked list.

To deal with multiple locks, consider the following questions:

- What do you want to lock?
- For multiple locks, in what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get one or more locks?

In managing record locks, you must plan a failure strategy for the case in which you cannot obtain all the required locks. It is because of contention for these records that you have decided to use record locking in the first place. Different programs might

- wait a certain amount of time, and try again
- end the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- a combination of the above

Look now at the example of inserting an entry into a doubly linked list. All the following examples assume that a record is declared as follows:

```
struct record {
.../* data portion of record */...
    long prev;    /* index to previous record in the list */
    long next;    /* index to next record in the list */
};
```

For the example, assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be promoted to a write lock so that the record may be edited. Example 7-5 shows a function that can be used for this.

Example 7-5. Record Locking With Promotion Using fcntl()

```
/*
|| This function is called with a file descriptor and the
|| offsets to three records in it: this, here, and next.
|| The caller is assumed to hold read locks on both here and next.
|| This function promotes these locks to write locks.
|| If write locks on "here" and "next" are obtained
||    Set a write lock on "this".
||    Return index to "this" record.
|| If any write lock is not obtained:
||    Restore read locks on "here" and "next".
||    Remove all other locks.
||    Return -1.
*/
long set3Locks(int fd, long this, long here, long next)
{
    struct flock lck;
    lck.l_type = F_WRLCK;    /* setting a write lock */
    lck.l_whence = 0;        /* offsets are absolute */
    lck.l_len = sizeof(struct record);
    /* Promote the lock on "here" to write lock */
    lck.l_start = here;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        return (-1);
    }
    /* Lock "this" with write lock */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Failed to lock "this"; return "here" to read lock. */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLKW, &lck);
        return (-1);
    }
    /* Promote lock on "next" to write lock */
    lck.l_start = next;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Failed to promote "next"; return "here" to read lock... */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLK, &lck);
        /* ...and remove lock on "this".  */
        lck.l_type = F_UNLCK;
        lck.l_start = this;
        (void) fcntl(fd, F_SETLK, &lck);
        return (-1)
    }
    return (this);
}
```

Example 7-5 uses the F_SETLKW command to fcntl(), with the result that the calling process will sleep if there are conflicting locks at any of the three points. If the F_SETLK command was used instead, the fcntl() system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections (as in Example 7-2).

It is possible to unlock or change the type of lock on a subsection of a previously set lock; this may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Example 7-6 shows a similar example using the lockf() function. Since it does not support read locks, all (write) locks are referenced generically as locks.

Example 7-6. Record Locking Using lockf()

```
/*
|| This function is called with a file descriptor and the
|| offsets to three records in it: this, here, and next.
|| The caller is assumed to hold no locks on any of the records.
|| This function tries to lock "here" and "next" using lockf().
|| If locks on "here" and "next" are obtained
||    Set a lock on "this".
||    Return index to "this" record.
|| If any lock is not obtained:
||    Remove all other locks.
||   Return -1.
*/
long set3Locks(int fd, long this, long here, long next)
{
   /* Set a lock on "here" */
   (void) lseek(fd, here, 0);
   if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
      return (-1);
   }
   /* Lock "this" */
   (void) lseek(fd, this, 0);
   if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
      /* Failed to lock "this"; clear "here" lock. */
      (void) lseek(fd, here, 0);
      (void) lockf(fd, F_ULOCK, sizeof(struct record));
      return (-1);
   }
   /* Lock "next" */
   (void) lseek(fd, next, 0);
   if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
      /* Failed to lock "next"; release "here"... */
      (void) lseek(fd, here, 0);
      (void) lockf(fd, F_ULOCK, sizeof(struct record));
      /* ...and remove lock on "this".  */
      (void) lseek(fd, this, 0);
      (void) lockf(fd, F_ULOCK, sizeof(struct record));
      return (-1)
   }
   return (this);
}
```

Locks are removed in the same manner as they are set; only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process. An unlock can affect only locks that were placed by the unlocking process.

Getting Lock Information

You can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. To find this information, set up a lock as in the previous examples and use the F_GETLK command in the fcntl() call. If the lock passed to fcntl() would be blocked, the first blocking lock is returned to the process through the structure passed to fcntl(). That is, the lock data passed to fcntl() is overwritten by blocking lock information.

The returned information includes two pieces of data, *l_pidf* and *l_sysid*, that are used only with F_GETLK. These fields uniquely identify the process holding the lock. (For systems that do not support a distributed architecture, the value in *l_sysid* can be ignored.)

If a lock passed to fcntl() using the F_GETLK command is not blocked by another lock, the *l_type* field is changed to F_UNLCK and the remaining fields in the structure are unaffected.

Example 7-7 shows how to use this capability to print all the records locked by other processes. Note that if several read locks occur over the same record, only one of these is found.

Example 7-7. Detecting Contending Locks Using fcntl()

```
/*
|| This function takes a file descriptor and prints a report showing
|| all locks currently set on that file. The loop variable is the
|| l_start field of the flock structure. The function asks fcntl()
|| for the first lock that would block a lock from l_start to the end
|| of the file (l_len==0). When no lock would block such a lock,
|| the returned l_type contains F_UNLCK and the loop ends.
|| Otherwise the contending lock is displayed, l_start is set to
|| the end-point of that lock, and the loop repeats.
*/
void printAllLocksOn(int fd)
{
   struct flock lck;
   /* Find and print "write lock" blocked segments of file. */
   (void) printf("sysid pid type start length\n");
   lck.l_whence = 0;
   lck.l_start = 0L;
   lck.l_len = 0L;
   for( lck.l_type = 0; lck.l_type != F_UNLCK; )
   {
      lck.l_type = F_WRLCK;
      (void) fcntl(fd, F_GETLK, &lck);
      if (lck.l_type != F_UNLCK)
      {
         (void) printf("%5d %5d %c %8d %8d\n",
                    lck.l_sysid,
                    lck.l_pid,
                    (lck.l_type == F_WRLCK) ? 'W' : 'R',
                    lck.l_start,
                    lck.l_len);
         if (lck.l_len == 0)
            break; /* this lock goes to end of file, stop */
         lck.l_start += lck.l_len;
      }
```

```
    }
}
```

fcntl() with the F_GETLK command always returns correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The lockf() function with the F_TEST command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. Example 7-8 shows a code fragment that uses lockf() to test for a lock on a file.

Example 7-8. Testing for Contending Lock Using lockf()

```
/* find a blocked record. */
/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
    switch (errno) {
    case EACCES:
    case EAGAIN:
        (void) printf("file is locked by another process\n");
        break;
    case EBADF:
        /* bad argument passed to lockf */
        perror("lockf");
        break;
    default:
        (void) printf("lockf: unknown error <%d>\n", errno);
        break;
    }
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent seeks to a point in the file, the child's file pointer is also set to that location. Similarly, when a share group of processes is created using sproc(), and the sproc() flag PR_SFDS is used to keep the open-file table synchronized for all processes (see the sproc(2) reference page), then there is a single file pointer for each file and it is shared by every process in the share group.

This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, in lockf() at all times and in fcntl() when using an *l_whence* value of 1. Since there is no way to perform the sequence *lseek(); fcntl();* as an atomic operation, there is an obvious potential for race conditions—a lock might be set using a file pointer that was just changed by another process.

The solution is to have the child process close and reopen the file. This creates a distinct file descriptor for the use of that process. Another solution is to always use the fcntl()function for locking with an *l_whence* value of 0 or 2. This makes the locking function independent of the file pointer (processes might still contend for the use of the file pointer for other purposes such as direct-access input).

Deadlock Handling

A certain level of deadlock detection and avoidance is built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard lockf() call. This deadlock detection is valid only for processes that are locking files or records on a single system.

Deadlocks can potentially occur only when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call fails and sets *errno* to the deadlock error number.

If a process wishes to avoid using the system's deadlock detection, it should set its locks using F_GETLK instead of F_GETLKW.

Enforcing Mandatory Locking

File locking is usually an in-memory service of the IRIX kernel. The kernel keeps a table of locks that have been placed. Processes anywhere in the system update the table by calling fcntl() or lockf() to request locks. When all processes that use a file do this, and respect the results, file integrity can be maintained.

It is possible to extend file locking by making it mandatory on all processes, whether or not they were designed to be part of the cooperating group. Mandatory locking is enforced by the file I/O function calls. As a result, an independent process that calls write() to update a locked record is blocked or receives an error code.

The write() and other system functions test for a contending lock on a file that has mandatory locking applied. The test is made for every operation on that file. When the caller is a process that is cooperating in the lock, and has already set an appropriate lock, the mandatory test is unnecessary overhead.

Mandatory locking is enforced on a file-by-file basis, triggered by a bit in the file inode that is set by chmod (see the chmod(1) and chmod(2) reference pages). In order to enforce mandatory locking on a particular file, turn on the set-group-ID bit along with a nonexecutable group permission, as in these examples, which are equivalent:

```
$ chmod 2644 target.file
$ chmod +l target.file
```

The bit must be set before the file is opened; a change has no effect on a file that is already open.

Example 7-9 shows a fragment of code that sets mandatory lock mode on a given filename.

Example 7-9. Setting Mandatory Locking Permission Bits

```
#include <sys/types.h>
#include <sys/stat.h>
int setMandatoryLocking(char *filename)
{
  int mode;
  struct stat buf;
  if (stat(filename, &buf) < 0)
  {
    perror("stat(2)");
    return error;
  }
  mode = buf.st_mode;
  /* ensure group execute permission 0010 bit is off */
  mode &= ~(S_IEXEC>>3);
  /* turn on 'set group id bit' in mode */
  mode |= S_ISGID;
  if (chmod(filename, mode) < 0)
  {
    perror("chmod(2)");
    return error;
  }
  return 0;
}
```

When IRIX opens a file, it checks to see whether both of two conditions are true:

- Set-group-ID bit is 1.
- Group execute permission is 0.

When both are true, the file is marked for mandatory locking, and each use of creat(), open(), read(), and write() tests for contending locks.

Some points to remember about mandatory locking:

- Mandatory locking does not protect against file truncation with the truncate() function (see the truncate(2) reference page), which does not look for locks on the truncated portion of the file.
- Mandatory locking protects only those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX system file permissions.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.