R N S INSTITUTE OF TECHNOLOGY

CHANNASANDRA, BANGALORE - 98

UNIX SYSTEM PROGRAMMING

NOTES FOR 6TH SEMESTER INFORMATION SCIENCE
SUBJECT CODE: 10CS62

PREPARED BY

RAJKUMAR
Assistant Professor
Department of Information Science

DIVYA K
1RN09IS016
Information Science and Engineering
Divya.1rn09is016@gmail.com

Text Books: 1 Terrence Chan: Unix System Programming Using C++, Prentice Hall India, 1999.

2 W. Richard Stevens, Stephen A. Rago: Advanced Programming in the UNIX Environment, 2nd Edition, Pearson Education / PHI, 2005

Contents:

UNIT 1
INTRODUCTION

UNIX AND ANSI STANDARDS

UNIX is a computer operating system originally developed in 1969 by a group of AT&T employees at Bell Labs, including **Ken Thompson, Dennis Ritchie, Douglas McElroy and Joe Ossanna.** Today UNIX systems are split into various branches, developed over time by AT&T as well as various commercial vendors and non-profit organizations.

The ANSI C Standard

In 1989, **American National Standard Institute** (ANSI) proposed C programming language standard X3.159-1989 to standardise the language constructs and libraries. This is termed as ANSI C standard. This attempt to unify the implementation of the C language supported on all computer system.
The major differences between ANSI C and K&R C [Kernighan and Ritchie] are as follows:

- Function prototyping
- Support of the const and volatile data type qualifiers.
- Support wide characters and internationalization.
- Permit function pointers to be used without dereferencing.

Function prototyping

ANSI C adopts C++ function prototype technique where function definition and declaration include function names, arguments' data types, and return value data types. This enables ANSI C compilers to check for function calls in user programs that pass invalid number of arguments or incompatible arguments' data type.
These fix a major weakness of K&R C compilers: invalid function calls in user programs often pass compilation but cause programs to crash when they are executed.

Eg:
```
unsigned long foo(char * fmt, double data)
{
        /*body of foo*/
}
```
External declaration of this function foo is
```
unsigned long foo(char * fmt, double data);
```

eg:
```
int printf(const char* fmt,...........);
```
specify variable number of arguments

Support of the const and volatile data type qualifiers.

- The **const** keyword declares that some data cannot be changed.
  Eg: `int printf(const char* fmt,...........);`
  Declares a fmt argument that is of a const char * data type, meaning that the function printf cannot modify data in any character array that is passed as an actual argument value to fmt.
- **Volatile** keyword specifies that the values of some variables may change asynchronously, giving an hint to the compiler's optimization algorithm not to remove any "redundant" statements that involve "volatile" objects.

eg:
```
char get_io()
{
        volatile char* io_port = 0x7777;
        char ch = *io_port;        /*read first byte of data*/
        ch = *io_port;                    /*read second byte of data*/
}
```

If io_port variable is not declared to be volatile when the program is compiled, the compiler may eliminate second ch = *io_port statement, as it is considered redundant with respect to the previous statement.

- The **const** and **volatile** data type qualifiers are also supported in C++.

Support wide characters and internationalisation

- ANSI C supports internationalisation by allowing C-program to use wide characters. Wide characters use more than one byte of storage per character.
- ANSI C defines the **setlocale** function, which allows users to specify the format of date, monetary and real number representations.
  For eg: most countries display the date in dd/mm/yyyy format whereas US displays it in mm/dd/yyyy format.
- Function prototype of setlocale function is:

```
#include<locale.h>
char setlocale (int category, const char* locale);
```

- The setlocale function prototype and possible values of the category argument are declared in the <locale.h> header. The category values specify what format class(es) is to be changed.
- Some of the possible values of the category argument are:

category value		**effect on standard C functions/macros**
LC_CTYPE	⇒	Affects behavior of the <ctype.h> macros
LC_TIME	⇒	Affects date and time format.
LC_NUMERIC	⇒	Affects number representation format
LC_MONETARY	⇒	Affects monetary values format
LC_ALL	⇒	combines the affect of all above

Permit function pointers without dereferencing

ANSI C specifies that a function pointer may be used like a function name. No referencing is needed when calling a function whose address is contained in the pointer.
For Example, the following statement given below defines a function pointer funptr, which contains the address of the function foo.

```
extern void foo(double xyz,const int *ptr);
void (*funptr)(double,const int *)=foo;
```

The function foo may be invoked by either directly calling foo or via the funptr.

```
foo(12.78,"Hello world");
funptr(12.78,"Hello world");
```

K&R C requires funptr be dereferenced to call foo.

```
(* funptr) (13.48,"Hello usp");
```

ANSI C also defines a set of C processor(cpp) symbols, which may be used in user programs. These symbols are assigned actual values at compilation time.

cpp SYMBOL	**USE**
STDC	Feature test macro. Value is 1 if a compiler is ANSI C, 0 otherwise
LINE	Evaluated to the physical line number of a source file.
FILE	Value is the file name of a module that contains this symbol.
DATE	Value is the date that a module containing this symbol is compiled.
TIME	value is the time that a module containing this symbol is compiled.

The following **test_ansi_c.c** program illustrates the use of these symbols:

```
#include<stdio.h>
int main()
{
        #if _STDC_==0
                printf("cc is not ANSI C compliant");
        #else
                printf("%s compiled at %s:%s.
                        This statement is at line %d\n",
                        _FILE_ , _DATE_ , _TIME_ , _LINE_ );
        #endif
                Return 0;
}
```

- Finally, ANSI C defines a set of standard library function & associated headers. These headers are the subset of the C libraries available on most system that implement K&R C.

The ANSI/ISO C++ Standard

These compilers support C++ classes, derived classes, virtual functions, operator overloading. Furthermore, they should also support template classes, template functions, exception handling and the iostream library classes.

Differences between ANSI C and C++

ANSI C	C++
Uses K&R C default function declaration for any functions that are referred before their declaration in the program.	Requires that all functions must be declared / defined before they can be referenced.
int foo(); ANSI C treats this as old C function declaration & interprets it as declared in following manner. int foo(........); → meaning that foo may be called with any number of arguments.	int foo(); C++ treats this as int foo(void); Meaning that foo may not accept any arguments.
Does not employ type_safe linkage technique and does not catch user errors.	Encrypts external function names for type_safe linkage. Thus reports any user errors.

The POSIX standards

- POSIX or "Portable Operating System Interface" is the name of a family of related standards specified by the IEEE to define the application-programming interface (API), along with shell and utilities interface for the software compatible with variants of the UNIX operating system.
- Because many versions of UNIX exist today and each of them provides its own set of API functions, it is difficult for system developers to create applications that can be easily ported to different versions of UNIX.
- Some of the subgroups of POSIX are POSIX.1, POSIX.1b & POSIX.1c are concerned with the development of set of standards for system developers.
- **POSIX.1**
  - ➢ This committee proposes a standard for a base operating system API; this standard specifies APIs for the manipulating of files and processes.
  - ➢ It is formally known as IEEE standard 1003.1-1990 and it was also adopted by the ISO as the international standard ISO/IEC 9945:1:1990.
- **POSIX.1b**
  - ➢ This committee proposes a set of standard APIs for a real time OS interface; these include IPC (inter-process communication).
  - ➢ This standard is formally known as IEEE standard 1003.4-1993.
- **POSIX.1c**
  - ➢ This standard specifies multi-threaded programming interface. This is the newest POSIX standard.
  - ➢ These standards are proposed for a generic OS that is not necessarily be UNIX system.
  - ➢ E.g.: VMS from Digital Equipment Corporation, OS/2 from IBM, & Windows NT from Microsoft Corporation are POSIX-compliant, yet they are not UNIX systems.
  - ➢ To ensure a user program conforms to POSIX.1 standard, the user should either define the manifested constant _POSIX_SOURCE at the beginning of each source module of the program (before inclusion of any header) as;

    ```
    #define _POSIX_SOURCE
    ```

    Or specify the -D_POSIX_SOURCE option to a C++ compiler (CC) in a compilation;

    ```
    %     CC -D_POSIX_SOURCE *.C
    ```
  - ➢ POSIX.1b defines different manifested constant to check conformance of user program to that standard. The new macro is _POSIX_C_SOURCE and its value indicates POSIX version to which a user program conforms. Its value can be:

_POSIX_C_SOURCE VALUES	MEANING
198808L	First version of POSIX.1 compliance
199009L	Second version of POSIX.1 compliance
199309L	POSIX.1 and POSIX.1b compliance

- _POSIX_C_SOURCE may be used in place of _POSIX_SOURCE. However, some systems that support POSIX.1 only may not accept the _POSIX_C_SOURCE definition.
- There is also a _POSIX_VERSION constant defined in <unistd.h> header. It contains the POSIX version to which the system conforms.

Program to check and display _POSIX_VERSION constant of the system on which it is run

```
#define  _POSIX_SOURCE
#define  _POSIX_C_SOURCE   199309L
#include<iostream.h>
#include<unistd.h>
int main()
{
#ifdef _POSIX_VERSION
       cout<<"System conforms to POSIX"<<"_POSIX_VERSION<<endl;
#else
       cout<<"_POSIX_VERSION undefined\n";
#endif
       return 0;
}
```

The POSIX Environment

Although POSIX was developed on UNIX, a POSIX complaint system is not necessarily a UNIX system. A few UNIX conventions have different meanings according to the POSIX standards. Most C and C++ header files are stored under the /usr/include directory in any UNIX system and each of them is referenced by

`#include<header-file-name>`

This method is adopted in POSIX. There need not be a physical file of that name existing on a POSIX conforming system.

The POSIX Feature Test Macros

POSIX.1 defines a set of feature test macro's which if defined on a system, means that the system has implemented the corresponding features. All these test macros are defined in **<unistd.h>** header.

Feature test macro	Effects if defined
_POSIX_JOB_CONTROL	The system supports the BSD style job control.
_POSIX_SAVED_IDS	Each process running on the system keeps the saved set UID and the set-GID, so that they can change its effective user-ID and group-ID to those values via seteuid and setegid API's.
_POSIX_CHOWN_RESTRICTED	If the defined value is -1, users may change ownership of files owned by them, otherwise only users with special privilege may change ownership of any file on the system.
_POSIX_NO_TRUNC	If the defined value is -1, any long pathname passed to an API is silently truncated to NAME_MAX bytes, otherwise error is generated.
_POSIX_VDISABLE	If defined value is -1, there is no disabling character for special characters for all terminal device files. Otherwise the value is the disabling character value.

Program to print POSIX defined configuration options supported on any given system.

```
/* show_test_macros.C */
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE     199309L
#include<iostream.h>
#include<unistd.h>
int main()
{
#ifdef _POSIX_JOB_CONTROL
       cout<<"system supports job control";
#else
       cout<<" system does not support job control\n";
#endif

#ifdef _POSIX_SAVED_IDS
       cout<<" system supports saved set-UID and set-GID";
#else
       cout<<" system does not support set-uid and gid\n";
#endif

#ifdef _POSIX_CHOWN_RESTRICTED
       cout<<"chown_restricted option is :"
              <<_POSIX_CHOWN_RESTRICTED<<endl;
#else
       cout<<"system does not support"
              <<" chown_restricted option\n";
#endif

#ifdef _POSIX_NO_TRUNC
       cout<<"pathname trunc option is:"
              << _POSIX_NO_TRUNC<<endl;
#else
       cout<<" system does not support system-wide pathname"
      <<"trunc option\n";
#endif

#ifdef _POSIX_VDISABLE
       cout<<"disable char. for terminal files is:"
              <<_POSIX_VDISABLE<<endl;
#else
       cout<<" system does not support _POSIX_VDISABLE \n";
#endif
       return 0;
}
```

Limits checking at Compile time and at Run time

POSIX.1 and POSIX.1b defines a set of system configuration limits in the form of manifested constants in the <limits.h> header.

The following is a list of POSIX.1 – defined constants in the <limits.h> header.

Compile time limit	Min. Value	Meaning
_POSIX_CHILD_MAX	6	Maximum number of child processes that may be created at any one time by a process.
_POSIX_OPEN_MAX	16	Maximum number of files that a process can open simultaneously.
_POSIX_STREAM_MAX	8	Maximum number of I/O streams opened by a process

		simultaneously.
_POSIX_ARG_MAX	4096	Maximum size, in bytes of arguments that may be passed to an *exec* function.
_POSIX_NGROUP_MAX	0	Maximum number of supplemental groups to which a process may belong
_POSIX_PATH_MAX	255	Maximum number of characters allowed in a path name
_POSIX_NAME_MAX	14	Maximum number of characters allowed in a file name
_POSIX_LINK_MAX	8	Maximum number of links a file may have
_POSIX_PIPE_BUF	512	Maximum size of a block of data that may be atomically read from or written to a pipe
_POSIX_MAX_INPUT	255	Maximum capacity of a terminal's input queue (bytes)
_POSIX_MAX_CANON	255	Maximum size of a terminal's canonical input queue
_POSIX_SSIZE_MAX	32767	Maximum value that can be stored in a ssize_t-typed object
_POSIX_TZNAME_MAX	3	Maximum number of characters in a time zone name

The following is a list of POSIX.1b – defined constants:

Compile time limit	Min. Value	Meaning
_POSIX_AIO_MAX	1	Number of simultaneous asynchronous I/O.
_POSIX_AIO_LISTIO_MAX	2	Maximum number of operations in one listio.
_POSIX_TIMER_MAX	32	Maximum number of timers that can be used simultaneously by a process.
_POSIX_DELAYTIMER_MAX	32	Maximum number of overruns allowed per timer.
_POSIX_MQ_OPEN_MAX	2	Maximum number of message queues that may be accessed simultaneously per process
_POSIX_MQ_PRIO_MAX	2	Maximum number of message priorities that can be assigned to the messages
_POSIX_RTSIG_MAX	8	Maximum number of real time signals.
_POSIX_SIGQUEUE_MAX	32	Maximum number of real time signals that a process may queue at any time.
_POSIX_SEM_NSEMS_MAX	256	Maximum number of semaphores that may be used simultaneously per process.
_POSIX_SEM_VALUE_MAX	32767	Maximum value that may be assigned to a semaphore.

Prototypes:

```
#include<unistd.h>

long sysconf(const int limit_name);
long pathconf(const char *pathname, int flimit_name);
long fpathconf(const int fd, int flimit_name);
```

The *limit_name* argument value is a manifested constant as defined in the <unistd.h> header. The possible values and the corresponding data returned by the *sysconf* function are:

Limit value	*Sysconf* return data
_SC_ARG_MAX	Maximum size of argument values (in bytes) that may be passed to an exec API call
_SC_CHILD_MAX	Maximum number of child processes that may be owned by a process simultaneously
_SC_OPEN_MAX	Maximum number of opened files per process

_SC_NGROUPS_MAX	Maximum number of supplemental groups per process
_SC_CLK_TCK	The number of clock ticks per second
_SC_JOB_CONTROL	The _POSIX_JOB_CONTROL value
_SC_SAVED_IDS	The _POSIX_SAVED_IDS value
_SC_VERSION	The _POSIX_VERSION value
_SC_TIMERS	The _POSIX_TIMERS value
_SC_DELAYTIMERS_MAX	Maximum number of overruns allowed per timer
_SC_RTSIG_MAX	Maximum number of real time signals.
_SC_MQ_OPEN_MAX	Maximum number of messages queues per process.
_SC_MQ_PRIO_MAX	Maximum priority value assignable to a message
_SC_SEM_MSEMS_MAX	Maximum number of semaphores per process
_SC_SEM_VALUE_MAX	Maximum value assignable to a semaphore.
_SC_SIGQUEUE_MAX	Maximum number of real time signals that a process may queue at any one time
_SC_AIO_LISTIO_MAX	Maximum number of operations in one listio.
_SC_AIO_MAX	Number of simultaneous asynchronous I/O.

All constants used as a sysconf argument value have the _SC prefix. Similarly the flimit_name argument value is a manifested constant defined by the <unistd.h> header. These constants all have the _PC_ prefix.
Following is the list of some of the constants and their corresponding return values from either pathconf or fpathconf functions for a named file object.

Limit value	*Pathconf* return data
_PC_CHOWN_RESTRICTED	The _POSIX_CHOWN_RESTRICTED value
_PC_NO_TRUNC	Returns the _POSIX_NO_TRUNC value
_PC_VDISABLE	Returns the _POSIX_VDISABLE value
_PC_PATH_MAX	Maximum length of a pathname (in bytes)
_PC_NAME_MAX	Maximum length of a filename (in bytes)
_PC_LINK_MAX	Maximum number of links a file may have
_PC_PIPE_BUF	Maximum size of a block of data that may be read from or written to a pipe
_PC_MAX_CANON	maximum size of a terminal's canonical input queue
_PC_MAX_INPUT	Maximum capacity of a terminal's input queue.

These variables may be used at compile time, such as the following:

```
char pathname [ _POSIX_PATH_MAX + 1 ];
for (int i=0; i < _POSIX_OPEN_MAX; i++)
      close(i);          //close all file descriptors
```

The following *test_config.C* illustrates the use of *sysconf, pathcong and fpathconf:*

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE    199309L
#include<stdio.h>
#include<iostream.h>
#include<unistd.h>

int main()
{
      int res;
      if((res=sysconf(_SC_OPEN_MAX))==-1)
            perror("sysconf");
      else
            cout<<"OPEN_MAX:"<<res<<endl;
```

```
        if((res=pathconf("/",_PC_PATH_MAX))==-1)
                perror("pathconf");
        else
                cout<<"max path name:"<<(res+1)<<endl;
        if((res=fpathconf(0,_PC_CHOWN_RESTRICTED))==-1)
                perror("fpathconf");
        else
                cout<<"chown_restricted for stdin:"<<res<<endl;
        return 0;
}
```

The POSIX.1 FIPS Standard

FIPS stands for Federal Information Processing Standard. The FIPS standard is a restriction of the POSIx.1 – 1988 standard, and it requires the following features to be implemented in all FIPS-conforming systems:

- Job control
- Saved set-UID and saved set-GID
- Long path name is not supported
- The _POSIX_CHOWN_RESTRICTED must be defined
- The _POSIX_VDISABLE symbol must be defined
- The NGROUP_MAX symbol's value must be at least 8
- The read and write API should return the number of bytes that have been transferred after the APIs have been interrupted by signals
- The group ID of a newly created file must inherit the group ID of its containing directory.

The FIPS standard is a more restrictive version of the POSIX.1 standard

The X/OPEN Standards

The X/Open organization was formed by a group of European companies to propose a common operating system interface for their computer systems. The portability guides specify a set of common facilities and C application program interface functions to be provided on all UNIX based open systems. In 1973, a group of computer vendors initiated a project called "*common open software environment*" (COSE). The goal of the project was to define a single UNIX programming interface specification that would be supported by all type vendors. The applications that conform to ANSI C and POSIX also conform to the X/Open standards but not necessarily vice-versa.

UNIX AND POSIX APIs

API → A set of application programming interface functions that can be called by user programs to perform system specific functions.

Most UNIX systems provide a common set of API's to perform the following functions.

- Determine the system configuration and user information.
- Files manipulation.
- Processes creation and control.
- Inter-process communication.
- Signals and daemons
- Network communication.

The POSIX APIs

In general POSIX API's uses and behaviours' are similar to those of Unix API's. However, user's programs should define the _POSIX_SOURCE or _POSIX_C_SOURCE in their programs to enable the POSIX API's declaration in header files that they include.

The UNIX and POSIX Development Environment

POSIX provides portability at the source level. This means that you transport your source program to the target machine, compile it with the standard C compiler using conforming headers and link it with the standard libraries.

Some commonly used POSIX.1 and UNIX API's are declared in <unistd.h> header. Most of POSIX.1, POSIX>1b and UNIX API object code is stored in the libc.a and lib.so libraries.

API Common Characteristics

- Many APIs returns an **integer value** which indicates the termination status of their execution
- API **return -1** to indicate the **execution has failed**, and the global variable **errno** is set with an error code.
- a user process may call **perror** function to print a diagnostic message of the failure to the std o/p, or it may call **strerror** function and gives it errno as the actual argument value; the strerror function returns a diagnostic message string and the user process may print that message in its preferred way
- the possible error status codes that may be assigned to errno by any API are defined in the <errno.h> header.

Following is a list of commonly occur error status codes and their meanings:

Error status code	Meaning
EACCESS	A process does not have access permission to perform an operation via a API.
EPERM	A API was aborted because the calling process does not have the superuser privilege.
ENOENT	An invalid filename was specified to an API.
BADF	A API was called with invalid file descriptor.
EINTR	A API execution was aborted due to a signal interruption
EAGAIN	A API was aborted because some system resource it requested was temporarily unavailable. The API should be called again later.
ENOMEM	A API was aborted because it could not allocate dynamic memory.
EIO	I/O error occurred in a API execution.
EPIPE	A API attempted to write data to a pipe which has no reader.
EFAULT	A API was passed an invalid address in one of its argument.
ENOEXEC	A API could not execute a program via one of the exec API
ECHILD	A process does not have any child process which it can wait on.

UNIT 2
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.

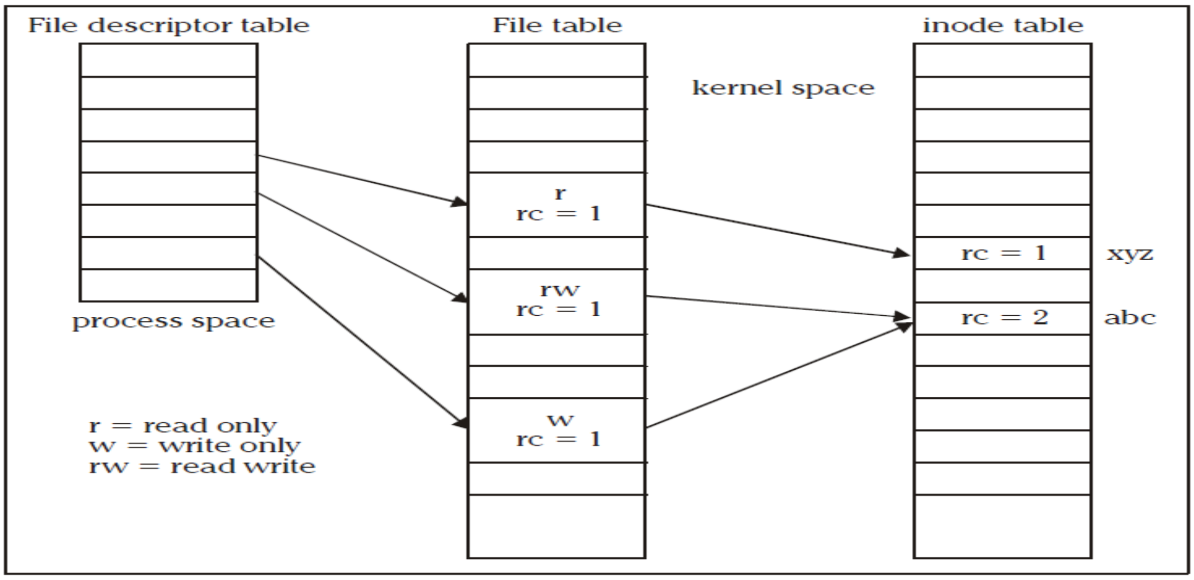
2. The kernel will scan the file table in its kernel space to find an unused entry that can be assigned to reference the file.

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.

Relationship of C Stream Pointers and File Descriptors

The major difference between the stream pointer and the file descriptors are as follows:

Stream pointer	FILE descriptor

Stream pointers are allocated via the fopen function call.
Eg: FILE *fp;
fp=fopen(&&);
	File descriptor are allocated via the open system call
	Eg: int fd;
	fd=open(&..);

Stream pointer is efficient to use for application doing extensive read from or write to files.	File descriptors are more efficient for applications that do frequent random access of file
Stream pointers is supported on all operating system such as VMS,CMS,DOS and UNIX that provide C compilers	File pointers are used only in UNIX and POSIX 1 compliant systems

The file descriptor associated with a stream pointer can be extracted by *fileno* macro, which is declared in the <stdio.h> header.

```
int fileno(FILE * stream_pointer);
```

To convert a file descriptor to a stream pointer, we can use *fdopen* C library function

```
FILE *fdopen(int file_descriptor, char * open_mode);
```

The following lists some C library functions and the underlying UNIX APIs theyuse to perform their functions:

C library function	UNIX system call used
fopen	**open**
fread, fgetc, fscanf, fgets	**read**
fwrite, fputc, fprintf, fputs	**write**
fseek, fputc, fprintf, fputs	**lseek**
fclose	**close**

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

UNIT 3
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:
  - o If a file is opened for read-only, lseek will fail.
  - o If a file is opened for write access, lseek will succeed.
  - o 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:
  - o The first one is the user name to be assigned to files
  - o 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 File 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

The following list_dir.C program illustrates the uses of the mkdir, opendir, readdir, closedir and rmdir APIs:

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

int main(int agc, char* argv[])
{
Dirent* dp;
DIR* dir_fdesc;
while(--argc>0)
{
if(!(dir_fdesc=opendir(*++argv)))
{
if(mkdir(*argv,S_IRWXU | S_IRWXG | S_IRWXO)==-1)
perror("opendir");
continue;
}
for(int i=0;i<2;i++)
for(int cnt=0;dp=readdir(dir_fdesc);)
{
if(i)
cout<<dp->d_name<<endl;
if(strcmp(dp->d_name,".") && strcmp(dp->d_name,".."))
cnt++;
}
if(!cnt)
{
rmdir(*argv);
break;
}
rewinddir(dir_fdesc);
}
closedir(dir_fdesc);
}
}
```

Device file APIs

- Device files are used to interface physical device with application programs.
- A process with superuser privileges to create a device file must call the mknod API.
- The user ID and group ID attributes of a device file are assigned in the same manner as for regular files.
- When a process reads or writes to a device file, the kernel uses the major and minor device numbers of a file to select a device driver function to carry out the actual data transfer.
- Device file support is implementation dependent. UNIX System defines the mknod API to create device files.
- The prototype of mknod is

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

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

- The first argument pathname is the pathname of a device file to be created.
- The second argument mode specifies the access permission, for the owner, group and others, also S_IFCHR or S_IBLK flag to be assigned to the file.
- The third argument device_id contains the major and minor device number.
- **Example**
  ```
  mknod("SCSI5",S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO,(15<<8) | 3);
  ```
- The above function creates a block device file "divya", to which all the three i.e. read, write and execute permission is granted for user, group and others with major number as 8 and minor number 3.
- On success mknod API returns 0 , else it returns -1

The following test_mknod.C program illustrates the use of the mknod, open, read, write and close APIs on a block device file.

```
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>

int main(int argc, char* argv[])
{
	if(argc!=4)
	{
		cout<<"usage:"<<argv[0]<<"<file><major_no><minor_no>";
		return 0;
	}
	int major=atoi(argv[2],minor=atoi(argv[3]);
	(void) mknod(argv[1], S_IFCHR | S_IRWXU | S_IRWXG | S_IRWXO, (major<<8) | minor);

	int rc=1,fd=open(argv[1],O_RDW | O_NONBLOCK | O_NOCTTY);
	char buf[256];
	while(rc && fd!=-1)
	if((rc=read(fd,buf,sizeof(buf)))<0)
		perror("read");
```

```
        else if(rc)
                cout<<buf<<endl;
        close(fd);
}
```

FIFO file API's

- FIFO files are sometimes called named pipes.
- Pipes can be used only between related processes when a common ancestor has created the pipe.
- Creating a FIFO is similar to creating a file.
- Indeed the pathname for a FIFO exists in the file system.
- The prototype of mkfifo is

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

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

- The first argument pathname is the pathname(filename) of a FIFO file to be created.
- The second argument mode specifies the access permission for user, group and others and as well as the S_IFIFO flag to indicate that it is a FIFO file.
- On success it returns 0 and on failure it returns –1.
- **Example**

```
mkfifo("FIFO5",S_IFIFO | S_IRWXU | S_IRGRP | S_ROTH);
```

- The above statement creates a FIFO file "divya" with read-write-execute permission for user and only read permission for group and others.
- Once we have created a FIFO using mkfifo, we open it using open.
- Indeed, the normal file I/O functions (read, write, unlink etc) all work with FIFOs.
- When a process opens a FIFO file for reading, the kernel will block the process until there is another process that opens the same file for writing.
- Similarly whenever a process opens a FIFO file write, the kernel will block the process until another process opens the same FIFO for reading.
- This provides a means for synchronization in order to undergo inter-process communication.
- If a particular process tries to write something to a FIFO file that is full, then that process will be blocked until another process has read data from the FIFO to make space for the process to write.
- Similarly, if a process attempts to read data from an empty FIFO, the process will be blocked until another process writes data to the FIFO.
- From any of the above condition if the process doesn't want to get blocked then we should specify O_NONBLOCK in the open call to the FIFO file.
- If the data is not ready for read/write then open returns –1 instead of process getting blocked.
- If a process writes to a FIFO file that has no other process attached to it for read, the kernel will send SIGPIPE signal to the process to notify that it is an illegal operation.
- Another method to create FIFO files (not exactly) for inter-process communication is to use the pipe system call.
- The prototype of pipe is

```
#include <unistd.h>
int pipe(int fds[2]);
```

- Returns 0 on success and –1 on failure.
- If the pipe call executes successfully, the process can read from fd[0] and write to fd[1]. A single process with a pipe is not very useful. Usually a parent process uses pipes to communicate with its children.

The following test_fifo.C example illustrates the use of mkfifo, open, read, write and close APIs for a FIFO file:

```
#include<iostream.h>
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<string.h>
#include<errno.h>
```

```
int main(int argc,char* argv[])
{
        if(argc!=2 && argc!=3)
        {
                cout<<"usage:"<<argv[0]<<"<file> [<arg>]";
                return 0;
        }
        int fd;
        char buf[256];
        (void) mkfifo(argv[1], S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO );
        if(argc==2)
        {
                fd=open(argv[1],O_RDONLY | O_NONBLOCK);
                while(read(fd,buf,sizeof(buf))==-1 && errno==EAGAIN)
                        sleep(1);
                while(read(fd,buf,sizeof(buf))>0)
                        cout<<buf<<endl;
        }
        else
        {
                fd=open(argv[1],O_WRONLY);
                write(fd,argv[2],strlen(argv[2]));
        }
        close(fd);
}
```

Symbolic Link File API's

- A symbolic link is an indirect pointer to a file, unlike the hard links which pointed directly to the inode of the file.
- Symbolic links are developed to get around the limitations of hard links:
    - Symbolic links can link files across file systems.
    - Symbolic links can link directory files
    - Symbolic links always reference the latest version of the files to which they link
    - There are no file system limitations on a symbolic link and what it points to and anyone can create a symbolic link to a directory.
    - Symbolic links are typically used to move a file or an entire directory hierarchy to some other location on a system.
    - A symbolic link is created with the symlink.
    - The prototype is

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

int symlink(const char *org_link, const char *sym_link);
int readlink(const char* sym_link,char* buf,int size);
int lstat(const char * sym_link, struct stat* statv);
```

- The org_link and sym_link arguments to a sym_link call specify the original file path name and the symbolic link path name to be created.

```
/* Program to illustrate symlink function */
#include<unistd.h>
#include<sys/types.h>
#include<string.h>

int main(int argc, char *argv[])
{
        char *buf [256], tname [256];
        if (argc ==4)
        return symlink(argv[2], argv[3]); /* create a symbolic link */
        else
        return link(argv[1], argv[2]); /* creates a hard link */
}
```

General file class

- In C++, fstream class is used to define objects, which represent files in a file system.
- The fstream class contains member functions like open, close, read, write, seekg and tellg.
- The fstream class does not provide member functions like stat chown, chmod, utime and link functions on its object.
- To overcome the fstream class deficiency a new filebase class is defined, which incorporates the fstream class properties and additional function to allow users to get or change object file attributes and to create hard links.

```
/* filebase.h */
#define FILEBASE_H
#ifndef FILEBASE_H
#include<fstream.h>
#include<iostream.h>
#include<sys/types.h>
#include<string.h>
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<utime.h>
typedef enum
{
REG_FILE='r', DIR_FILE='d', CHAR_FILE='c' PIPE_FILE='p',
SYM_FILE = 's', BLK_FILE = 'b', UNKNOWN_FILE = '?'
}
FILE_TYPE_ENUM;
class filebase : public fstream
{
protected:
       char *filename;
       friend ostream& operator<<(ostream& os,filebase& fobj)
       {return os;};
public :
file base()    {filename =0:};
file base(const char *fn, int flags, int prot = filebuf :: openprot)
ifstream(fn, flags, prot)
{
filename = newchar[strlen(fn) + 1];
strcpy(filename, fn);
};
virtual ~filebase()  {delete filename;};
virtual int create(const char *fn, mode_t mode)
{
       return :: create(fn, mode);
};
int fileno()
{
       return rdbuf()->fd();
};
int chmod(mode_t mode)
{
       return :: chmod(filename, mode);
};
int chown(uid_t uid, gid_t gid)
{
       return :: chown(filename, uid, gid);
};
int link (const char *new_link)
{
       return :: link (filename, new_link);
};
int utime (const struct utim buf *timbuf_ptr)
{
       return :: utime (filename, timbuf_ptr);
};
virtual int remove()
{
       return :: unlink(filename );
};
FILE_TYPE_ENUM file_type()
{
struct stat statv;
if (stat(filename, &statv)==0)
switch (statv.st_mode & S_IFMT)
{
case S_IFREG : return REG_FILE : /* Regular file */
case S_IFIFO : return PIPE_FILE; /* block device file */
case S_IFCHR : return CHAR_FILE ; /* character device file */
case S_IFDIR : return DIR_FILE ; /* Directory file */
case S_IFLNK : return SYM_FILE ; /* Symbolic link file */
}
return UNKNOWN_FILE;
};
};
#endif /*filebase.h*/
```

```
/* Program to illustrate the use of the file base class */
#include "filebase.h"
int main()
{
filebase file("/usr/text/usp.doc", ios :: in); //define an object
cout <<file;                             //display file attributes
file.chown(20, 40);                      //change UID and GID
file.utime( );                                  //touch time stamp
file.link("/home/divya/hdlnk");          //create a hard link
file.remove();                                  //remove the old link
}
```

Regfile Class for Regular Files

- The regfile class is defined as a subclass of filebase.
- The filebase class encapsulates most of the properties and functions needed to represent regular file objects in UNIX systems except file locking.
- Thus, objects of regfile class can do all regular file operations permitted in filebase class include file locking functions.
- regfile::lock → set read lock or write lock.
- regfile::lockw → wrapper over the regfile::lock and it locks files in blocking mode.
- regfile::getlock → queries lock information for a region of a file associated with a regfile object .
- regfile::unlock → unlocks a region of a file associated with a regfile object.

```
/* Regfile.h*/
#define REGFILE_H
#ifndef REGFILE_H
#include "filebase.h"
Class regfile : public filebase
{
Public :
              regfile(const char *fn, int mode, int prot): filebase(fn, mode, prot)
              {};
              ~regfile()    {};
int lock(int lck_type, off_t len, int cmd = F_SETLK)
{
       struct flock flck;
       if ((lck_type & ios ::in) == ios :: n)
              flck.l_type = F_RDLCK;
       else if (lck_type & ios :: out) == ios :: out)
              flck.l_type=F_WRLCK;
       else return -1;
       flck.l_whence _SEEK_CUR;
       flck.l_Start = (off_t)O;
       flck.l_len = len;
       return fcntl(fileno(), cmd, & flck);
};
int lockw(int lck_type, off_t len)
```

```
{
       return lock(lck_type, len, F_SETLKW);
}
int unlock(off_t len)
{
       Struct flock flck;
       flck.l_type = F_UNLCK;
       flck.l_whence = SEEK_CUR;
       flck.l_start = (off_t)0;
       flck.l_len = len;
       return fcntl(fileno(), F_SETLK, & flck);
};
int getlock(int lck_type, off_t len, struct flock &flck)
{
       if((lck_type &ios :: in) == ios :: in)
              flck.l_type= F_RDLCK;
       else if ((lck_type &ios :: out) == ios :: out)
              flck.l_type = F_WRLCK;
       else return -1;
       flck.l_whence = SEEK_CUR;
       flck.l_start = (off_t)0;
       flck.l_len = len;
       return fcntl(fileno(), F_GETLK, &flck);
};
};
#end if
```

```
/* Program to illustrate the use of regfile class */
#include "regfile.h"
int main()
{
ifstream ifs("/etc/passwd");
char buf[256];
regfile rfile("foo", ios :: out | ios :: in);
rfile.lock(ios :: out, 0); //set write lock for entire file
while (ifs.getline(buf, 256))
rfile << buf<<endl;
rfile.seekg(o, ios:: beg); //set fp to beginning of the file
rfile.unlock(10); //unlock first 10bytes of file
rfile.remove(); //remove the file
}
```

dirfile class for Directory file

The dirfile class is defined to encapsulate all UNIX directory file functions. The dirfile class defines the *create, open, read, tellg, seekg, close and remove* functions that use the UNIX directory file specific API's. The dirfile class definition is:

```
/* dirfile.h*/
#ifndef DIRFILE_H
#define DIRFILE_H
#include<dirent.h>
#include<string.h>
class dirfile
{
char *filename;
DIR *dir_ptr;
public : dirfile(const char *fn)
     {
     dir_ptr = opendir(fn);
     filename = strdup(fn);
     };
     ~dirfile()
       {
       if (dir_ptr) close();
       delete(filename);
       };
       int close()
       { if (dir_ptr) closedir(dir_ptr); };
       int creat(const char *fn, mode_t prot)
       { return mkdir(fn,prot); };
```

```
        int open(const char *fn)
        {
        dir_ptr = opendir(fn);
        return dir_ptr ? 0 : -1;
        };
        int read(char *buf, int size)
        {
        struct dirent *dp = readdir(dir_ptr);
        if (dp)
                strncpy(buf, dp-> d_name, size);
        return dp ? strlen(dp-> d_name);
        };
        off_t tellg()
        { return telldir(dir_ptr); };
        void seekg(streampos ofs,seek_dir d)
        { seekdir(dir_ptr, ofs); };
        int remove()
        { return rmdir(filename); };
};
#end if
```

```
/* Program to illustrate the use of dirfile class */
#include "dirfile.h"
int main()
{
dirfile ndir, edir("/etc");       //create a dirfile object to /etc
ndir.create ("/usr/lock/dirf.ex");
char buf[256];
while(edir.read (buf, 256))       //echo files in the /etc dir
        cout << buf<<endl;
edir.close();                     //close a directory file
}
```

FIFO file Class

A FIFO file object differs from a filebase object in that a FIFO file is created, and the tellg and seekg functions are invalid for FIFO file objects. The following pipefile class encapsulates all the FIFO file type properties:

```
/* pipefile.h */
#ifndef PIPEFILE_H
#define PIPEFILE_H
#include "filebase.h"
class pipefile : public filebase
{
public :
        pipefile(const char *fn, int flags, int prot) : filebase(fn, flags, prot) {};
        int create(const char *fn, mode_t prot)
        { return mkfifo(fn, prot); };
        streampos tellg()
        { return (stream pos) -1 };
};
#end if
```

```
/* Program to illustrate the use of pipefile.h*/
#include "pipefile.h"
int main(int argc, char *argv[])
{
pipefile nfifo("FIFO", argc==1 ? ios :: in, ios :: out , 0755);
if (argc >1)                                  //writer process
{
Cout<<"writer process write : "<< argv[1]<<endl;
nfifo write(argv [1], strlen (argv[1])+1);    //write data to FIFO
}
Else                                                 //reader process
{
char buf [256];
nfifo.read(buf, 256);                                //read data from FIFO
cout<<"Read from FIFO"<< buf<<endl;
}
nfifo.close();                                       //close FIFO file
}
```

Device File Class

A device file object has most of the properties of a regular file object except in the way that the device file object is created. Also the *tellg, seekg, lock, lockw, unlock and getlock* functions are invalid for any character-based device file objects.

```
/*devfile.h*/
#define DEVFILE_H
#ifndef DEVFILE_H
#include "regfile.h"
class devfile : public regfile
{
public :
devfile(const char *fn, int flags, int prot): regfile(fn, flags, prot)
{};
int create (const char *fn, mode_t prot, int major_no, int minor_no, char type='c')
{
if (type =='c')
       return mknod(fn, S_IFCHR | prot, (major_no <<8) |minor_no);
else
       return mknod(fn, S_IFBLK | prot, (major_no << 8) | minor_no);
};
streampos tellg()
{
if (file_type() == CHAR_FILE)
       return (streampos)-1;
else
       return fstream :: tellg();
};
istream seekg(streampos ofs, seek_dir d)
{
if (file_type() ! = CHAR_FILE)
       fstream :: seekg(ofs, d);
return *this :
};
int lock (int lck_type, off_t len, int cmd = F_SETLK)
{
if (file_type() != CHAR_FILE)
       return regfile :: lock(lck_type, len, cmd);
else   return -1;
};
int lockw(int lck_type, off_t len)
{
if (file_type() ! = CHAR_FILE)
       return regfile :: lockw(lck_type, len);
else   return-1;
};
int unlockw(off_t len)
{
if(file_type() ! = CHAR_FILE)
       return regfile :: unlock(len);
else   return -1;
};
int getlock(int lck_type, off_t len, struct flock &flck)
{
if (file_type()! = CHAR_FILE)
       return regfile :: getlock(lck_type, len, flck);
else   return -1;
};
};
#end if
```

```
/* Program to illustrate the use of devfile class */
#include "devfile.h"
int main()
{
devfile ndev("/dev/tty", ios :: out, 0777);
ndev <<"this is a sample output string";
ndev.close();
}
```

Symbolic link file class

A symbolic link file object differs from a filebase object in the way it is created. Also, a new member function called ref_path is provided to depict the path name of a file to which the symbolic link object refers.

```
/* symfile.h*/
#ifndef SYMFILE_H
#define SYMFILE_H
#include "filebase.h"
/*a class to encapsulate UNIX symbolic link file objects' properties*/
class symfile : public filebase
{
public :
symfile() {};
~symfile () {};
int setlink(const char *old_link, const char *new_link)
{
filename = new char[strlen(new_link) +1];
strcpy(filename, new_link);
return symlink(old_link, new_link);
};
void open(int mode)
{ fstream :: open(filename, mode); };
const char *ref_path()
{
static char buf[256];
if (readlink(filename, buf, 256))
       return buf;
else   return (char *)-1;
};
};
#end if
```

```
/* Program to illustrate the use of symfile class */
#include "symfile.h"
int main ()
{
char buf [256];
symfile nsym;
nsym.setlink ("/usr/file/chap 10", "/usr/xyz/sym.lnk");
nsym.open(ios:in);
while(nsym.getline(buf, 256))
       cout<<buf<<endl;           //read /usr/file/chap 10
cout<<nsym.ref_path()<<endl;      //echo "/usr/file/chap 10"
nsym.close();                     //close the symbolic link file
}
```

File listing Program

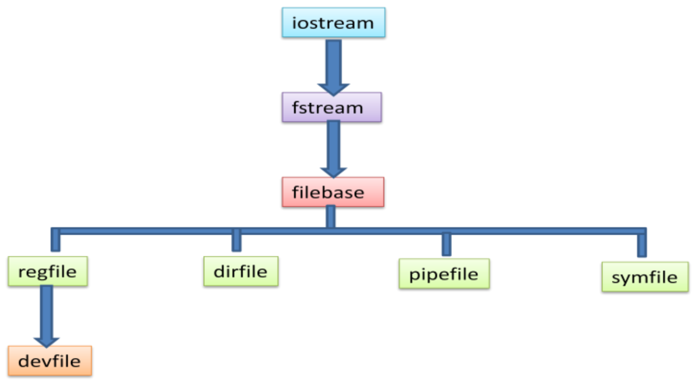
A C program re-implements the UNIX ls command to list file attributes of all path name arguments specified for the program using file base class and its subclasses. Further if an argument is a directory the program will list the file attributes of all files in that directory and any subdirectories. If a file is a symbolic link, the program will echo the pathname to which the link refers. Thus, the program behaves like the UNIX ls –lR command.

```
#include "filebase.h"
#include "dirfile.h"
#include "symfile.h"
void show_list(ostream &ofs, const char *fname, int deep);
extern void long_list (ostream &ofs, char *fn);
void show_dir(ostream &ofs, const char *fname)
{
dirfile dirobj(fname);
char buf[256];
ofs <<"Directory :" << fname;
while (dirobj.read(buf, 256))
{
       filebase fobj(buf, ios :: in, 0755);
       if(fobj.file_type==DIR_FILE)
```

```
              show_dir(ofs, buf);
       fobj.close();
}
dirobj.close();
}
void show_list(ostream &ofs, const char *fname, int deep)
{
long_list(ofs, fname);
filebase fobj(fname, ios::n, 0755);
if (fobj.file_type()==SYM_FILE)
{
       Symfile *symobj = (symfile *) fobj;
       ofs << "->" << symobj -> ref_path()<<endl;
}
else if (fobj.file_type() == DIR_FILE && deep)
       Show_dir(ofs, fname);
}
int main(int argc, char *argv[])
{
while (- -argc >0)          show_list(cout, *++argv, 1);
return 0;
}
```

Summary

The inheritance hierarchy of all the file classes defined in the chapter is:



UNIT 4
UNIX PROCESSES

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

main FUNCTION
A C program starts execution with a function called main. The prototype for the main function is

```
int main(int argc, char *argv[]);
```

where argc is the number of command-line arguments, and argv is an array of pointers to the arguments.

When a C program is executed by the kernel by one of the exec functions, a special start-up routine is called before the main function is called. The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler. This start-up routine takes values from the kernel, the command-line arguments and the environment and sets things up so that the main function is called.

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
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.

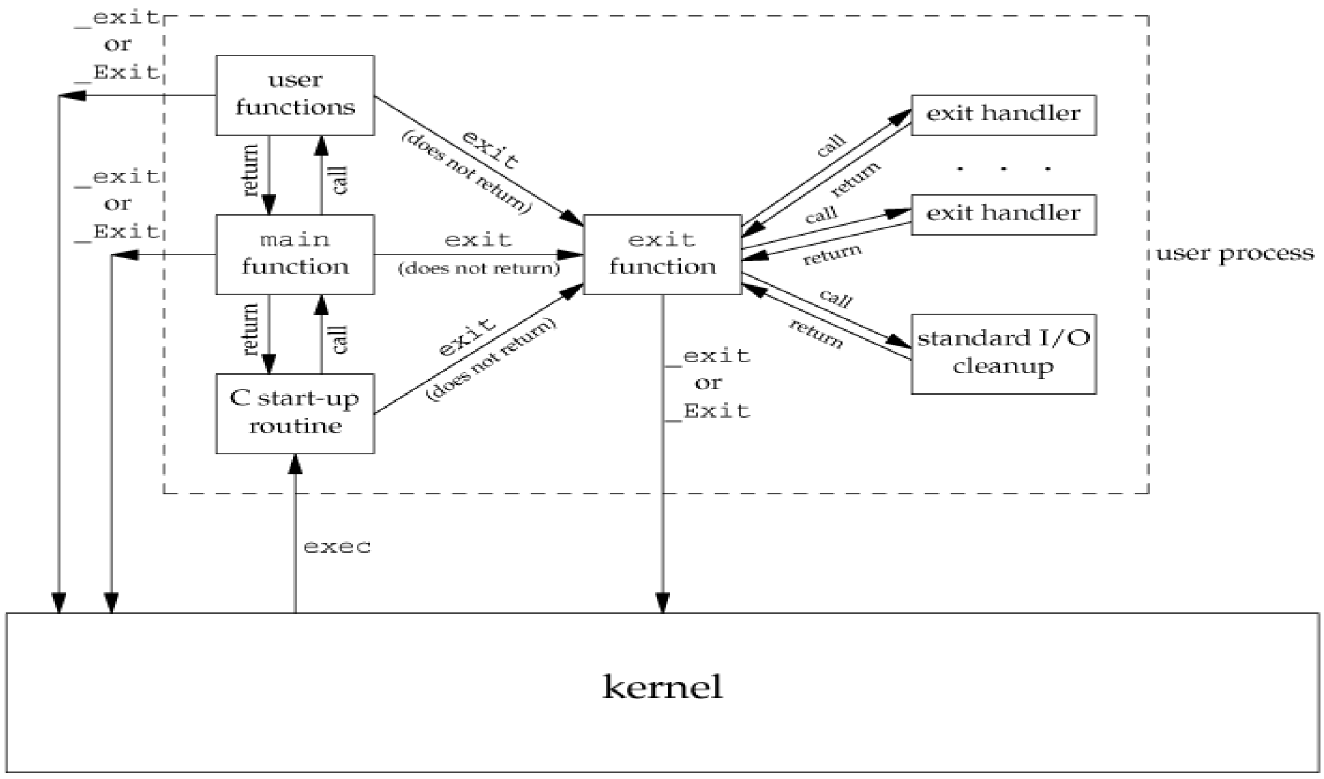
Example of exit handlers

```
#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.



COMMAND-LINE ARGUMENTS

When a program is executed, the process that does the `exec` can pass command-line arguments to the new program.

Example: Echo all command-line arguments to standard output

```
#include "apue.h"

int main(int argc, char *argv[])
{
    int     i;

    for (i = 0; i < argc;  i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

Output:

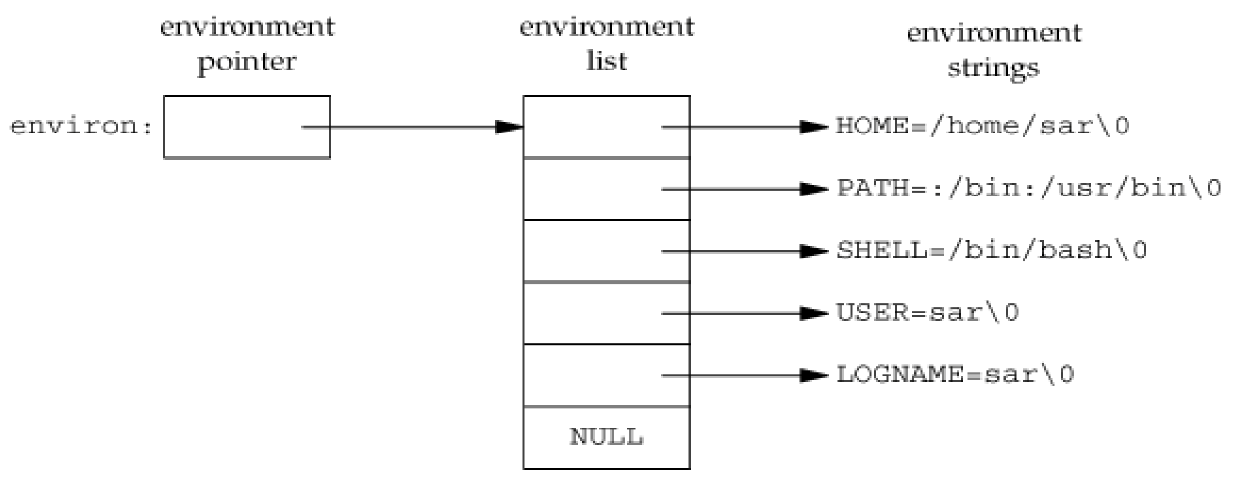
```
$ ./echoarg arg1 TEST foo
    argv[0]: ./echoarg
    argv[1]: arg1
    argv[2]: TEST
    argv[3]: foo
```

ENVIRONMENT LIST

Each program is also passed an environment list. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string. The address of the array of pointers is contained in the global variable `environ`:

```
    extern char **environ;
```

Figure : Environment consisting of five C character strings

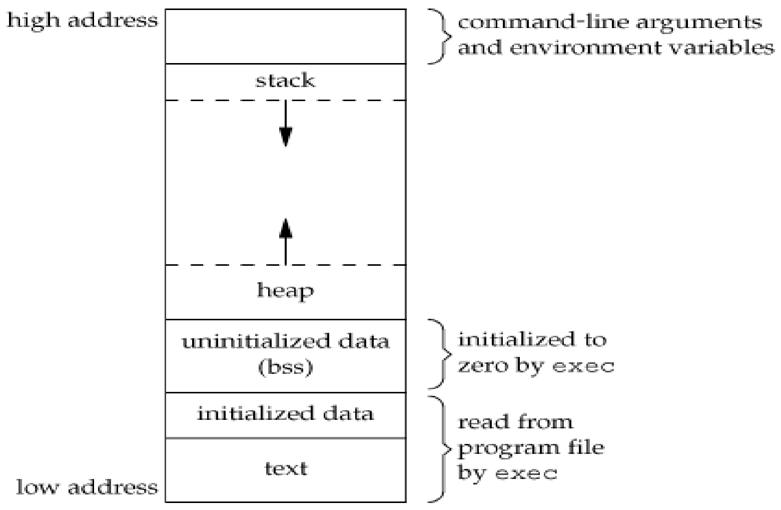


Generally any environmental variable is of the form: ***name=value.***

MEMORY LAYOUT OF A C PROGRAM

Historically, a C program has been composed of the following pieces:

- **Text segment**, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.
- **Initialized data segment**, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration
  ```
  int   maxcount = 99;
  ```
  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
- **Uninitialized data segment**, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration
  ```
  long  sum[1000];
  ```
  appearing outside any function causes this variable to be stored in the uninitialized data segment.
- **Stack**, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.
- **Heap**, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.


high address
stack
heap
uninitialized data (bss)
initialized data
text
low address
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec

SHARED LIBRARIES

Nowadays most UNIX systems support shared libraries. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference. This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called. Another advantage of shared libraries is that, library functions can be replaced with new versions without having to re-link, edit every program that uses the library. With cc compiler we can use the option –g to indicate that we are using shared library.

MEMORY ALLOCATION

ISO C specifies three functions for memory allocation:

- `malloc`, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
- `calloc`, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
- `realloc`, which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
```

All three return: non-null pointer if OK, `NULL` on error

```
void free(void *ptr);
```

The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. Because the three `alloc` functions return a generic `void *` pointer, if we `#include <stdlib.h>` (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type.

The function `free` causes the space pointed to by ptr to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three `alloc` functions.

The `realloc` function lets us increase or decrease the size of a previously allocated area. For example, if we allocate room for 512 elements in an array that we fill in at runtime but find that we need room for more than 512 elements, we can call `realloc`. If there is room beyond the end of the existing region for the requested space, then `realloc` doesn't have to move anything; it simply allocates the additional area at the end and returns the same pointer that we passed it. But if there isn't room at the end of the existing region, `realloc` allocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area.

The allocation routines are usually implemented with the `sbrk`(2) system call. Although `sbrk` can expand or contract the memory of a process, most versions of `malloc` and `free` never decrease their memory size. The space that we free is available for a later allocation, but the freed space is not usually returned to the kernel; that space is kept in the `malloc` pool.

It is important to realize that most implementations allocate a little more space than is requested and use the additional space for record keeping the size of the allocated block, a pointer to the next allocated block, and the like. This means that writing past the end of an allocated area could overwrite this record-keeping information in a later block. These types of errors are often catastrophic, but difficult to find, because the error may not show up until much later. Also, it is possible to overwrite this record keeping by writing before the start of the allocated area.

Because memory allocation errors are difficult to track down, some systems provide versions of these functions that do additional error checking every time one of the three `alloc` functions or `free` is called. These versions of the functions are often specified by including a special library for the link editor. There are also publicly available sources that you can compile with special flags to enable additional runtime checking.

Alternate Memory Allocators

Many replacements for `malloc` and `free` are available.

- **`libmalloc`**

SVR4-based systems, such as Solaris, include the `libmalloc` library, which provides a set of interfaces matching the ISO C memory allocation functions. The `libmalloc` library includes `mallopt`, a function that allows a process to set certain variables that control the operation of the storage allocator. A function called `mallinfo` is also available to provide statistics on the memory allocator.

- **`vmalloc`**

Vo describes a memory allocator that allows processes to allocate memory using different techniques for different regions of memory. In addition to the functions specific to `vmalloc`, the library also provides emulations of the ISO C memory allocation functions.

- **`quick-fit`**

Historically, the standard `malloc` algorithm used either a best-fit or a first-fit memory allocation strategy. Quick-fit is faster than either, but tends to use more memory. Free implementations of `malloc` and `free` based on quick-fit are readily available from several FTP sites.

- **`alloca` Function**

The function `alloca` has the same calling sequence as `malloc`; however, instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage is that we don't have to free the space; it goes away automatically when the function returns. The `alloca` function increases the size of the stack frame. The disadvantage is that some systems can't support `alloca`, if it's impossible to increase the size of the stack frame after the function has been called.

ENVIRONMENT VARIABLES

The environment strings are usually of the form: ***name=value.*** The UNIX kernel never looks at these strings; their interpretation is up to the various applications. The shells, for example, use numerous environment variables. Some, such as HOME and USER, are set automatically at login, and others are for us to set. We normally set environment variables in a shell start-up file to control the shell's actions. The functions that we can use to set and fetch values from the variables are setenv, putenv, and getenv functions. The prototype of these functions are

```
#include <stdlib.h>
char *getenv(const char *name);
```

Returns: pointer to value associated with name, NULL if not found.

Note that this function returns a pointer to the value of a **name=value** string. We should always use getenv to fetch a specific value from the environment, instead of accessing environ directly. In addition to fetching the value of an environment variable, sometimes we may want to set an environment variable. We may want to change the value of an existing variable or add a new variable to the environment. The prototypes of these functions are

```
#include <stdlib.h>
int putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

All return: 0 if OK, nonzero on error.

- The **putenv** function takes a string of the form **name=value** and places it in the environment list. If name already exists, its old definition is first removed.
- The **setenv** function sets name to value. If name already exists in the environment, then
  (a) if rewrite is nonzero, the existing definition for name is first removed;
  (b) if rewrite is 0, an existing definition for name is not removed, name is not set to the new value, and no error occurs.
- The **unsetenv** function removes any definition of name. It is not an error if such a definition does not exist.

Note the difference between **putenv** and **setenv**. Whereas **setenv** must allocate memory to create the **name=value** string from its arguments, **putenv** is free to place the string passed to it directly into the environment.

Environment variables defined in the Single UNIX Specification

Variable	Description
COLUMNS	terminal width
DATEMSK	getdate(3) template file pathname
HOME	home directory
LANG	name of locale
LC_ALL	name of locale
LC_COLLATE	name of locale for collation
LC_CTYPE	name of locale for character classification
LC_MESSAGES	name of locale for messages
LC_MONETARY	name of locale for monetary editing
LC_NUMERIC	name of locale for numeric editing
LC_TIME	name of locale for date/time formatting
LINES	terminal height
LOGNAME	login name
MSGVERB	fmtmsg(3) message components to process
NLSPATH	sequence of templates for message catalogs
PATH	list of path prefixes to search for executable file
PWD	absolute pathname of current working directory
SHELL	name of user's preferred shell
TERM	terminal type
TMPDIR	pathname of directory for creating temporary files
TZ	time zone information

NOTE:

- If we're modifying an existing name:
  - If the size of the new value is less than or equal to the size of the existing value, we can just copy the new string over the old string.
  - If the size of the new value is larger than the old one, however, we must `malloc` to obtain room for the new string, copy the new string to this area, and then replace the old pointer in the environment list for name with the pointer to this allocated area.
- If we're adding a new name, it's more complicated. First, we have to call `malloc` to allocate room for the name=value string and copy the string to this area.
  - Then, if it's the first time we've added a new name, we have to call `malloc` to obtain room for a new list of pointers. We copy the old environment list to this new area and store a pointer to the name=value string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally, we set `environ` to point to this new list of pointers.
  - If this isn't the first time we've added new strings to the environment list, then we know that we've already allocated room for the list on the heap, so we just call `realloc` to allocate room for one more pointer. The pointer to the new name=value string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

setjmp AND longjmp FUNCTIONS

In C, we can't `goto` a label that's in another function. Instead, we must use the `setjmp` and `longjmp` functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Returns: 0 if called directly, nonzero if returning from a call to `longjmp`

```
void longjmp(jmp_buf env, int val);
```

The setjmp function records or marks a location in a program code so that later when the longjmp function is called from some other function, the execution continues from the location onwards. The env variable(the first argument) records the necessary information needed to continue execution. The env is of the jmp_buf defined in <setjmp.h> file, it contains the task.

Example of setjmp and longjmp

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD    5
jmp_buf jmpbuffer;

int main(void)
{
     char    line[MAXLINE];

     if (setjmp(jmpbuffer) != 0)
         printf("error");
     while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
     exit(0);
}

 ...

void cmd_add(void)
{
    int   token;
    token = get_token();
    if (token < 0)    /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

- The setjmp function always returns ‘0’ on its success when it is called directly in a process (for the first time).
- The longjmp function is called to transfer a program flow to a location that was stored in the env argument.
- The program code marked by the env must be in a function that is among the callers of the current function.
- When the process is jumping to the target function, all the stack space used in the current function and its callers, upto the target function are discarded by the longjmp function.
- The process resumes execution by re-executing the setjmp statement in the target function that is marked by env. The return value of setjmp function is the value(val), as specified in the longjmp function call.
- The ‘val’ should be nonzero, so that it can be used to indicate where and why the longjmp function was invoked and process can do error handling accordingly.

Note: The values of ***automatic*** and ***register*** variables are indeterminate when the longjmp is called but static and global variable are unaltered. The variables that we don’t want to roll back after longjmp are declared with keyword ‘volatile’.

Figure 7.10. Stack frames after cmd_add has been called

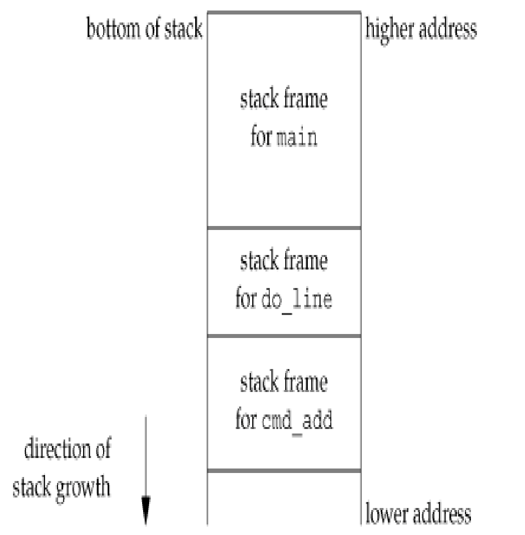
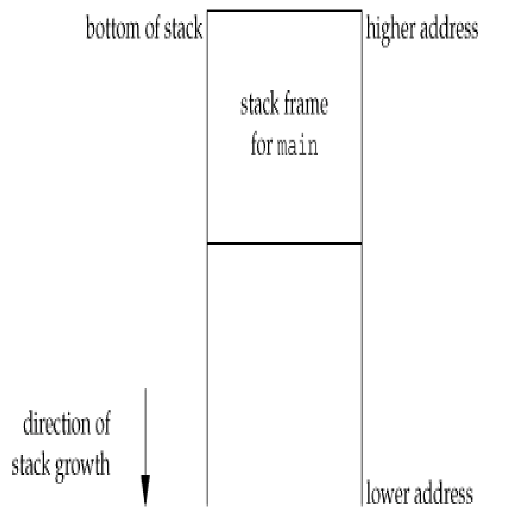


Figure 7.12. Stack frame after longjmp has been called



getrlimit AND setrlimit FUNCTIONS

Every process has a set of resource limits, some of which can be queried and changed by the geTRlimit and setrlimit functions.

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

int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit  *rlptr);
```

Both return: 0 if OK, nonzero on error

Each call to these two functions specifies a single resource and a pointer to the following structure:

```
struct rlimit
{
  rlim_t  rlim_cur;   /* soft limit: current limit */
  rlim_t  rlim_max;   /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

- A process can change its soft limit to a value less than or equal to its hard limit.
- A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
- Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant RLIM_INFINITY.

RLIMIT_AS	The maximum size in bytes of a process's total available memory.
RLIMIT_CORE	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
RLIMIT_CPU	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.
RLIMIT_DATA	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.
RLIMIT_FSIZE	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZ signal.
RLIMIT_LOCKS	The maximum number of file locks a process can hold.
RLIMIT_MEMLOCK	The maximum amount of memory in bytes that a process can lock into memory using mlock(2).
RLIMIT_NOFILE	The maximum number of open files per process. Changing this limit affects the value returned by the sysconf function for its _SC_OPEN_MAX argument
RLIMIT_NPROC	The maximum number of child processes per real user ID. Changing this limit affects the value returned for _SC_CHILD_MAX by the sysconf function
RLIMIT_RSS	Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.
RLIMIT_SBSIZE	The maximum size in bytes of socket buffers that a user can consume at any given time.
RLIMIT_STACK	The maximum size in bytes of the stack.
RLIMIT_VMEM	This is a synonym for RLIMIT_AS.

The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes.

Example: Print the current resource limits

```
#include "apue.h"
#if defined(BSD) || defined(MACOS)
#include <sys/time.h>
#define FMT "%10lld "
#else
#define FMT "%10ld "
#endif
#include <sys/resource.h>

#define doit(name) pr_limits(#name, name)
```

```
static void pr_limits(char *, int);

int main(void)
{

#ifdef  RLIMIT_AS
    doit(RLIMIT_AS);
#endif
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);
#ifdef  RLIMIT_LOCKS
    doit(RLIMIT_LOCKS);
#endif
#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
    doit(RLIMIT_NOFILE);
#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
#ifdef  RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
#endif
    doit(RLIMIT_STACK);
#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}

static void pr_limits(char *name, int resource)
{
    struct rlimit limit;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite) ");
    else
        printf(FMT, limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)");
    else
        printf(FMT, limit.rlim_max);
    putchar((int)'\n');
}
```

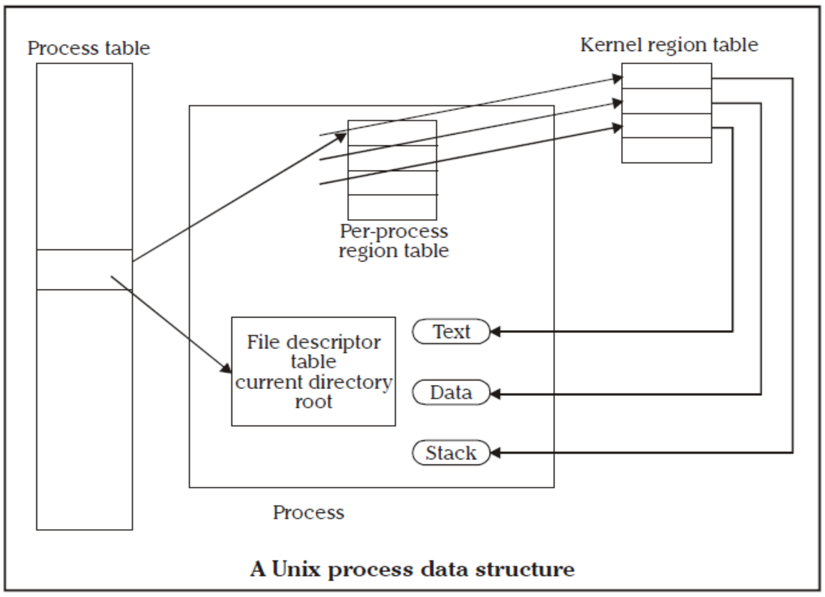
UNIX KERNEL SUPPORT FOR PROCESS

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.



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.

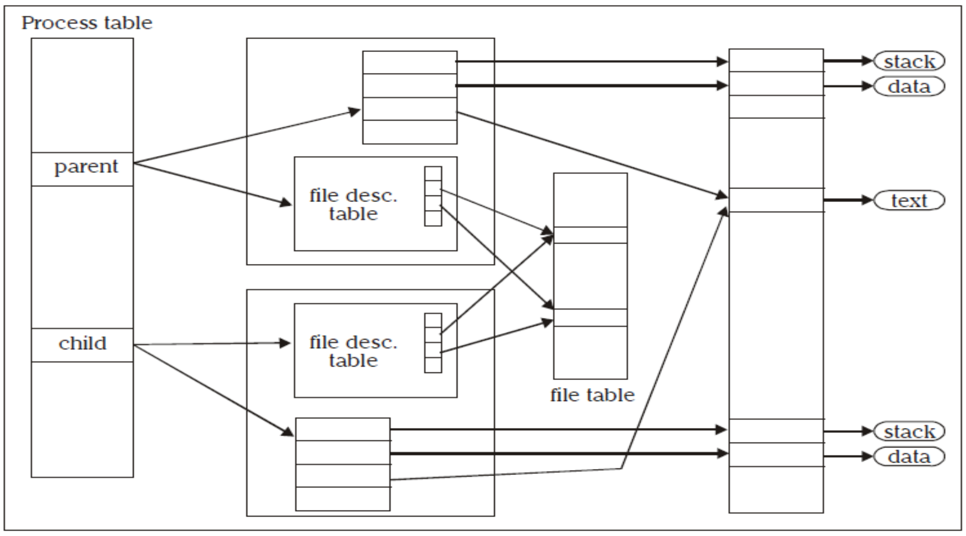


Figure: Parent & child relationship after fork

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.

UNIT 5
PROCESS CONTROL

INTRODUCTION

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

PROCESS IDENTIFIERS

```
#include <unistd.h>

pid_t getpid(void);
```

Returns: process ID of calling process

```
pid_t getppid(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

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. [*Also refer lab program 3.sh*]

File Sharing

Consider a process that has three different files opened for standard input, standard output, and standard error. On return from `fork`, we have the arrangement shown in Figure 8.2.

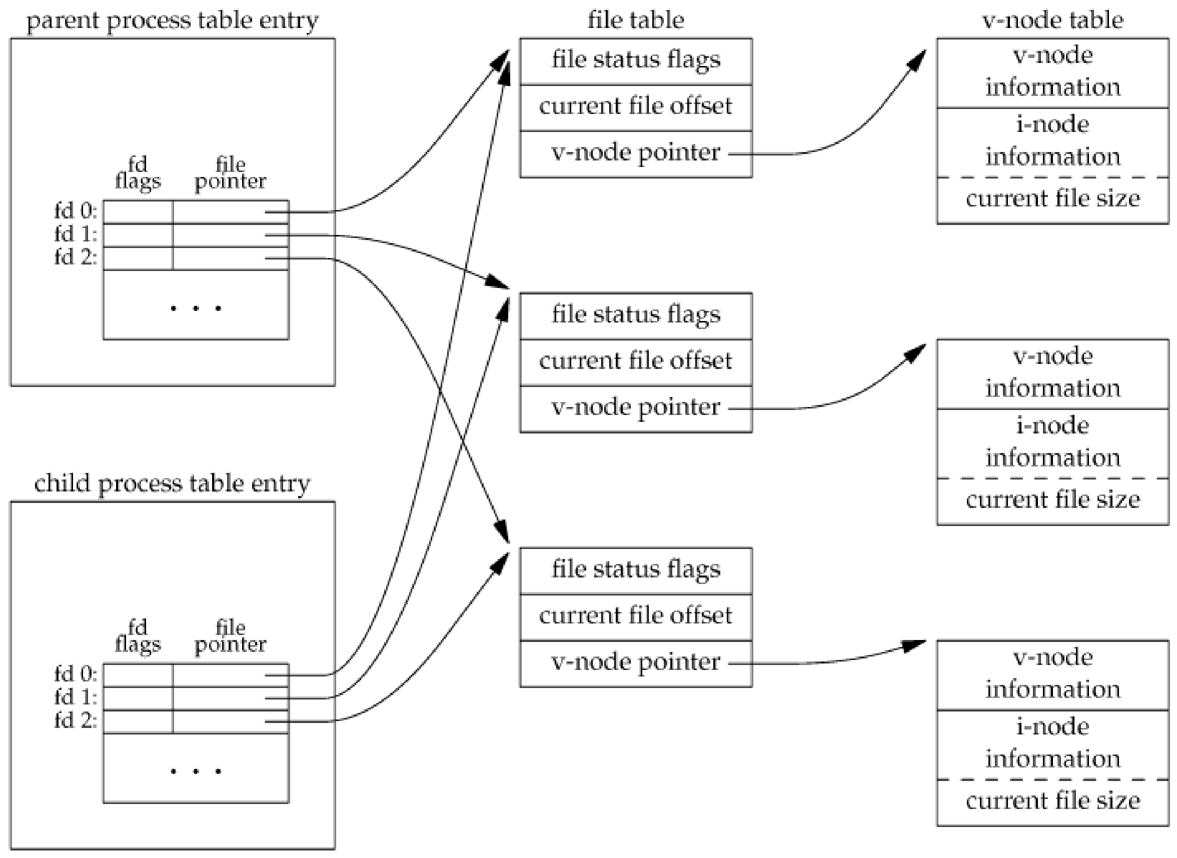


Figure 8.2 Sharing of open files between parent and child after fork

- It is important that the parent and the child share the same file offset.
- Consider a process that `fork`s a child, then `wait`s for the child to complete.
- Assume that both processes write to standard output as part of their normal processing.
- If the parent has its standard output redirected (by a shell, perhaps) it is essential that the parent's file offset be updated by the child when the child writes to standard output.
- In this case, the child can write to standard output while the parent is `wait`ing for it; on completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote.
- If the parent and the child did not share the same file offset, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.

There are two normal cases for handling the descriptors after a `fork`.

- ✓ The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
- ✓ Both the parent and the child go their own ways. Here, after the `fork`, the parent closes the descriptors that it doesn't need, and the child does the same thing. This way, neither interferes with the other's open descriptors. This scenario is often the case with network servers.

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

- o Real user ID, real group ID, effective user ID, effective group ID
- o Supplementary group IDs
- o Process group ID
- o 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) if too many processes are already in the system, which usually means that something else is wrong, or

(b) if the total number of processes for this real user ID exceeds the system's limit.

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.

Example of `vfork` function

```
#include "apue.h"
int     glob = 6;       /* external variable in initialized data */

int main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }
    /*
     * Parent continues here.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Output:

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

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.

wait AND waitpid FUNCTIONS

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, whereas `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 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.

Print a description of the `exit` status

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

Void pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

Program to Demonstrate various `exit` statuses

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

Int main(void)
{
    pid_t   pid;
    int     status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        abort();                    /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        status /= 0;                /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

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.

Macros to examine the termination status returned by `wait` and `waitpid`

Macro	Description
WIFEXITED(status)	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS` (status) to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`,or `_Exit`.
WIFSIGNALED (status)	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute `WTERMSIG` (status) to fetch the signal number that caused the termination. Additionally, some implementations (but not the Single UNIX Specification) define the macro `WCOREDUMP` (status) that returns true if a core file of the terminated process was generated.
WIFSTOPPED (status)	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG` (status) to fetch the signal number that caused the child to stop.
WIFCONTINUED (status)	True if status was returned for a child that has been continued after a job control stop

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.

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. We'll return to this feature when we discuss the `popen` function.
- ✓ 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.

Program to Avoid zombie processes by calling `fork` twice

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

Int main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */
        /*
         * We're the second child; our parent becomes init as soon
         * as our real parent calls exit() in the statement above.
         * Here's where we'd continue executing, knowing that when
         * we're done, init will reap our status.
         */
        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)  /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

Output:

```
$ ./a.out
$ second child, parent pid = 1
```

`waitid` FUNCTION

The `waitid` function is similar to `waitpid`, but provides extra flexibility.

```
#include <sys/wait.h>
Int waited(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns: 0 if OK, -1 on error

The *idtype* constants for waited are as follows:

Constant	Description
`P_PID`	Wait for a particular process: id contains the process ID of the child to wait for.
`P_PGID`	Wait for any child process in a particular process group: id contains the process group ID of the children to wait for.
`P_ALL`	Wait for any child process: id is ignored.

The options argument is a bitwise OR of the flags as shown below: these flags indicate which state changes the caller is interested in.

Constant	Description
WCONTINUED	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
WEXITED	Wait for processes that have exited.
WNOHANG	Return immediately instead of blocking if there is no child exit status available.
WNOWAIT	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to `wait`, `waitid`,or `waitpid`.
WSTOPPED	Wait for a process that has stopped and whose status has not yet been reported.

`wait3` AND `wait4` FUNCTIONS

The only feature provided by these two functions that isn't provided by the `wait`, `waitid`, and `waitpid` functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.
The prototypes of these functions are:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
pid_t wait3(int *statloc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

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

The resource information includes such statistics as the amount of user CPU time, the amount of system CPU time, number of page faults, number of signals received etc. the resource information is available only for terminated child process not for the process that were stopped due to job control.

RACE CONDITIONS

A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.

Example: The program below outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.

```
#include "apue.h"

static void charatatime(char *);

int main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

<u>Output</u>:

```
   $ ./a.out
   ooutput from child
   utput from parent
   $ ./a.out
   ooutput from child
   utput from parent
   $ ./a.out
   output from child
   output from parent
```

<u>program modification to avoid race condition</u>

```
#include "apue.h"

   static void charatatime(char *);

   int main(void)
   {
       pid_t   pid;

+      TELL_WAIT();
+
       if ((pid = fork()) < 0) {
           err_sys("fork error");
       } else if (pid == 0) {
+          WAIT_PARENT();      /* parent goes first */
           charatatime("output from child\n");
       } else {
           charatatime("output from parent\n");
+          TELL_CHILD(pid);
       }
       exit(0);
   }
   static void
```

```
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

When we run this program, the output is as we expect; there is no intermixing of output from the two processes.

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.

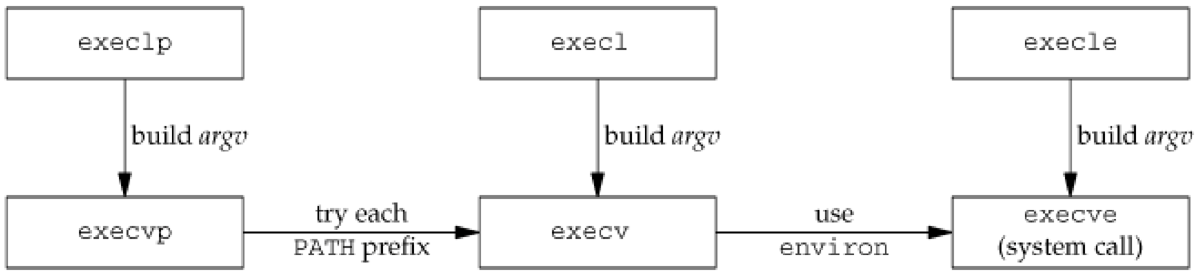
Function	pathname	filename	Arg list	argv[]	environ	envp[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(letter in name)		p	l	v		e

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:

- o 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`.



Relationship of the six exec functions

Example of `exec` functions

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

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

Output:

```
$ ./a.out
  argv[0]: echoall
  argv[1]: myarg1
  argv[2]: MY ARG2
  USER=unknown
```

```
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
                            47 more lines that aren't shown
HOME=/home/sar
```

Note that the shell prompt appeared before the printing of `argv[0]` from the second `exec`. This is because the parent did not `wait` for this child process to finish.

Echo all command-line arguments and all environment strings

```
#include "apue.h"

Int main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

CHANGING USER IDs AND GROUP IDs

When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access. Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

```
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

Both return: 0 if OK, 1 on error

There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

- If the process has superuser privileges, the `setuid` function sets the real user ID, effective user ID, and saved set-user-ID to uid.
- If the process does not have superuser privileges, but uid equals either the real user ID or the saved set-user-ID, `setuid` sets only the effective user ID to uid. The real user ID and the saved set-user-ID are not changed.
- If neither of these two conditions is true, `errno` is set to `EPERM`, and 1 is returned.

We can make a few statements about the three user IDs that the kernel maintains.

- Only a superuser process can change the real user ID. Normally, the real user ID is set by the `login`(1) program when we log in and never changes. Because `login` is a superuser process, it sets all three user IDs when it calls `setuid`.
- The effective user ID is set by the `exec` functions only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the `exec` functions leave the effective user ID as its current value. We can call `setuid` at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.
- The saved set-user-ID is copied from the effective user ID by `exec`. If the file's set-user-ID bit is set, this copy is saved after `exec` stores the effective user ID from the file's user ID.

ID	exec		setuid(uid)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged

				user
real user ID	unchanged	unchanged	set to uid	unchanged
effective user ID	unchanged	set from user ID of program file	set to uid	set to uid
saved set-user ID	copied from effective user ID	copied from effective user ID	set to uid	unchanged

The above figure summarises the various ways these three user IDs can be changed

setreuid and setregid Functions

Swapping of the real user ID and the effective user ID with the setreuid function.

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

Both return : 0 if OK, -1 on error

We can supply a value of 1 for any of the arguments to indicate that the corresponding ID should remain unchanged. The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user- ID operations.

seteuid and setegid functions :

POSIX.1 includes the two functions seteuid and setegid. These functions are similar to setuid and setgid, but only the effective user ID or effective group ID is changed.

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

Both return : 0 if OK, 1 on error

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged user, only the effective user ID is set to uid. (This differs from the setuid function, which changes all three user IDs.)

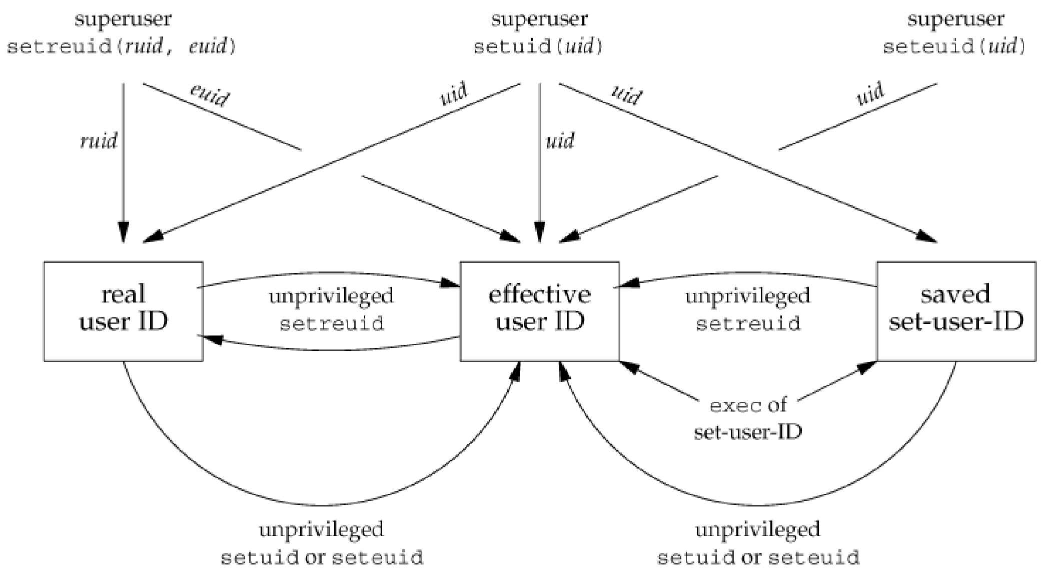


Figure: Summary of all the functions that set the various user Ids

INTERPRETER FILES

These files are text files that begin with a line of the form

```
    #! pathname [ optional-argument ]
```

The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line

```
   #!/bin/sh
```

The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., PATH is not used). The recognition of these files is done within the kernel as part of processing the exec system call. The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file. Be sure to differentiate between the interpreter filea text file that begins with #!and the interpreter, which is specified by the pathname on the first line of the interpreter file.
Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the #!, the pathname, the optional argument, the terminating newline, and any spaces.

A program that execs an interpreter file

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

Int main(void)
{
    pid_t   pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0) /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Output:

```
   $ cat /home/sar/bin/testinterp
   #!/home/sar/bin/echoarg foo
   $ ./a.out
   argv[0]: /home/sar/bin/echoarg
   argv[1]: foo
   argv[2]: /home/sar/bin/testinterp
   argv[3]: myarg1
   argv[4]: MY ARG2
```

system FUNCTION

```
#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.
Because 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 other than EINTR, 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.

Program: The system function, without signal handling

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

Int system(const char *cmdstring)    /* version without signal handling */
{
    pid_t   pid;
    int     status;
    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

Program: Calling the `system` function

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

Int main(void)
{
    int      status;

    if ((status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

   if ((status = system("nosuchcommand")) < 0)
       err_sys("system() error");
   pr_exit(status);

   if ((status = system("who; exit 44")) < 0)
       err_sys("system() error");
   pr_exit(status);

   exit(0);
}
```

Program: Execute the command-line argument using `system`

```
#include "apue.h"

Int main(int argc, char *argv[])
{
    int     status;

    if (argc < 2)
        err_quit("command-line argument required");

    if ((status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Program: Print real and effective user IDs

```
#include "apue.h"

int
main(void)
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

PROCESS ACCOUNTING

- Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates.
- These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on.
- A superuser executes accton with a pathname argument to enable accounting.
- The accounting records are written to the specified file, which is usually /var/account/acct. Accounting is turned off by executing accton without any arguments.
- The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork.
- Each accounting record is written when the process terminates.
- This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started.
- The accounting records correspond to processes, not programs.
- A new record is initialized by the kernel for the child after a fork, not when a new program is executed.

The structure of the accounting records is defined in the header <sys/acct.h> and looks something like

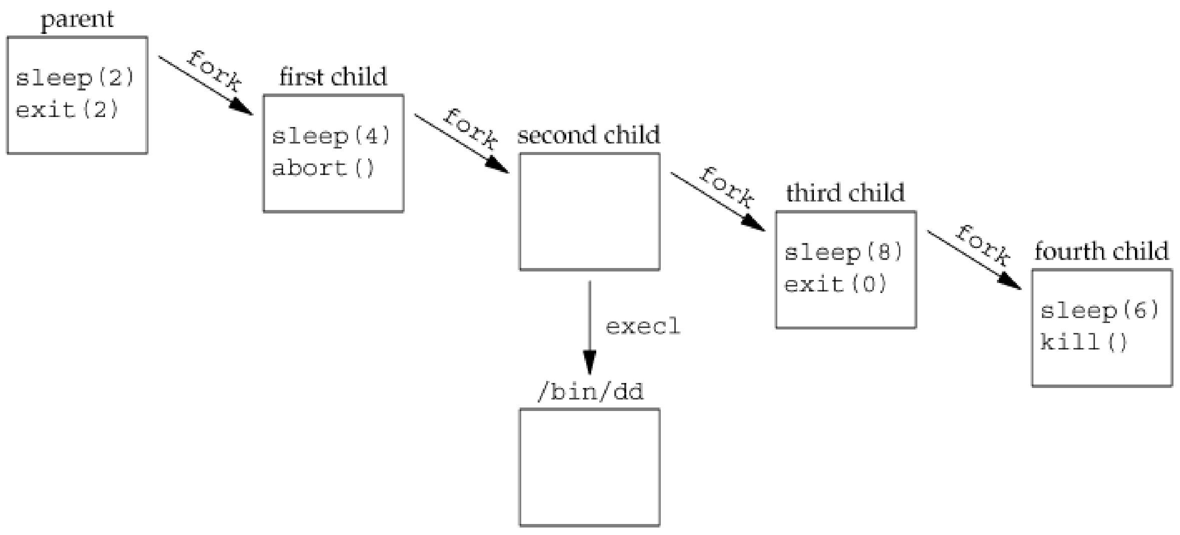
```
typedef  u_short comp_t;   /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char   ac_flag;      /* flag  */
  char   ac_stat;      /* termination status (signal & core flag only) */
                       /* (Solaris only) */
  uid_t  ac_uid;       /* real user ID */
  gid_t  ac_gid;       /* real group ID */
  dev_t  ac_tty;       /* controlling terminal */
  time_t ac_btime;     /* starting calendar time */
  comp_t ac_utime;     /* user CPU time (clock ticks) */
  comp_t ac_stime;     /* system CPU time (clock ticks) */
  comp_t ac_etime;     /* elapsed time (clock ticks) */
  comp_t ac_mem;       /* average memory usage */
  comp_t ac_io;        /* bytes transferred (by read and write) */
                       /* "blocks" on BSD systems */
  comp_t ac_rw;        /* blocks read or written */
                       /* (not present on BSD systems) */
  char   ac_comm[8];   /* command name: [8] for Solaris, */
                       /* [10] for Mac OS X, [16] for FreeBSD, and */
                       /* [17] for Linux */
};
```

Values for ac_flag from accounting record

ac_flag	Description
AFORK	process is the result of fork, but never called exec
ASU	process used superuser privileges
ACOMPAT	process used compatibility mode
ACORE	process dumped core
AXSIG	process was killed by a signal
AEXPND	expanded accounting entry

Program to generate accounting data

```
#include "apue.h"

Int main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* parent */
        sleep(2);
        exit(2);                /* terminate with exit status 2 */
    }

                                /* first child */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(4);
        abort();                /* terminate with core dump */
    }

                                /* second child */
   if ((pid = fork()) < 0)
       err_sys("fork error");
   else if (pid != 0) {
       execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
       exit(7);                 /* shouldn't get here */
   }

                                /* third child */
   if ((pid = fork()) < 0)
       err_sys("fork error");
   else if (pid != 0) {
       sleep(8);
       exit(0);                 /* normal exit */
   }

                                /* fourth child */
   sleep(6);
   kill(getpid(), SIGKILL);     /* terminate w/signal, no core dump */
   exit(6);                     /* shouldn't get here */
}
```



Process structure for accounting example

USER IDENTIFICATION

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call `getpwuid(getuid())`, but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in and the `getlogin` function provides a way to fetch that login name.

```
#include <unistd.h>

char *getlogin(void);
```

Returns : pointer to string giving login name if OK, NULL on error

This function can fail if the process is not attached to a terminal that a user logged in to.

PROCESS TIMES

We describe three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the `times` function to obtain these values for itself and any terminated children.

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

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, 1 on error

This function fills in the `tms` structure pointed to by buf:

```
struct tms {
  clock_t  tms_utime;  /* user CPU time */
  clock_t  tms_stime;  /* system CPU time */
  clock_t  tms_cutime; /* user CPU time, terminated children */
  clock_t  tms_cstime; /* system CPU time, terminated children */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value.

PROCESS RELATIONSHIPS

INTRODUCTION

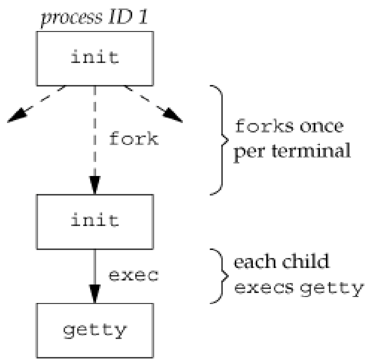
In this chapter, we'll look at process groups in more detail and the concept of sessions that was introduced by POSIX.1. We'll also look at the relationship between the login shell that is invoked for us when we log in and all the processes that we start from our login shell.

TERMINAL LOGINS

The terminals were either local (directly connected) or remote (connected through a modem). In either case, these logins came through a terminal device driver in the kernel.

The system administrator creates a file, usually `/etc/ttys`, that has one line per terminal device. Each line specifies the name of the device and other parameters that are passed to the `getty` program. One parameter is the baud rate of the terminal, for example. When the system is bootstrapped, the kernel creates process ID 1, the `init` process, and it is `init` that brings the system up multiuser. The `init` process reads the file `/etc/ttys` and, for every terminal device that allows a login, does a `fork` followed by an `exec` of the program `getty`. This gives us the processes shown in Figure 9.1.

Figure 9.1. Processes invoked by `init` to allow terminal logins



All the processes shown in Figure 9.1 have a real user ID of 0 and an effective user ID of 0 (i.e., they all have superuser privileges). The `init` process also `exec`s the `getty` program with an empty environment.

It is `getty` that calls `open` for the terminal device. The terminal is opened for reading and writing. If the device is a modem, the `open` may delay inside the device driver until the modem is dialed and the call is answered. Once the device is open, file descriptors 0, 1, and 2 are set to the device. Then `getty` outputs something like `login:` and waits for us to enter our user name.

When we enter our user name, `getty`'s job is complete, and it then invokes the `login` program, similar to

```
execle("/bin/login", "login", "-p", username, (char *)0, envp);
```

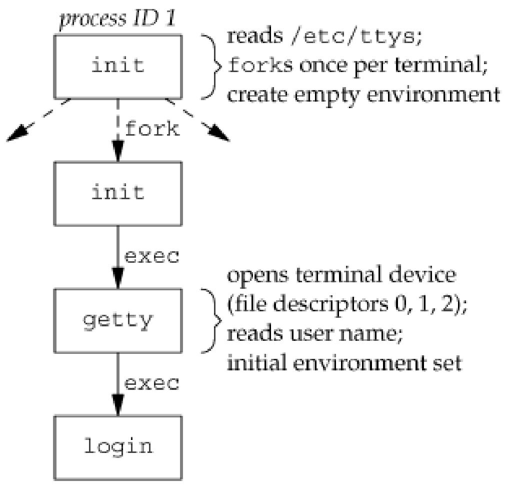

process ID 1
init
reads /etc/ttys; forks once per terminal; create empty environment
fork
init
exec
getty
opens terminal device (file descriptors 0, 1, 2); reads user name; initial environment set
exec
login

Figure 9.2. State of processes after `login` has been invoked

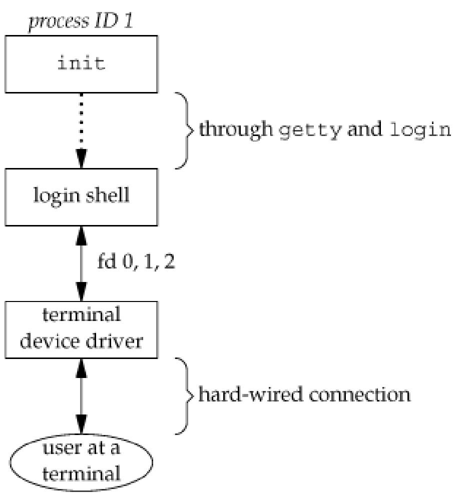
All the processes shown in Figure 9.2 have superuser privileges, since the original `init` process has superuser privileges.

If we log in correctly, `login` will

- Change to our home directory (`chdir`)
- Change the ownership of our terminal device (`chown`) so we own it
- Change the access permissions for our terminal device so we have permission to read from and write to it
- Set our group IDs by calling `setgid` and `initgroups`
- Initialize the environment with all the information that `login` has: our home directory (`HOME`), shell (`SHELL`), user name (`USER` and `LOGNAME`), and a default path (`PATH`)
- Change to our user ID (`setuid`) and invoke our login shell, as in **`execl("/bin/sh", "-sh", (char *)0);`**

The minus sign as the first character of `argv[0]` is a flag to all the shells that they are being invoked as a login shell. The shells can look at this character and modify their start-up accordingly.

Figure 9.3. Arrangement of processes after everything is set for a terminal login



NETWORK LOGINS

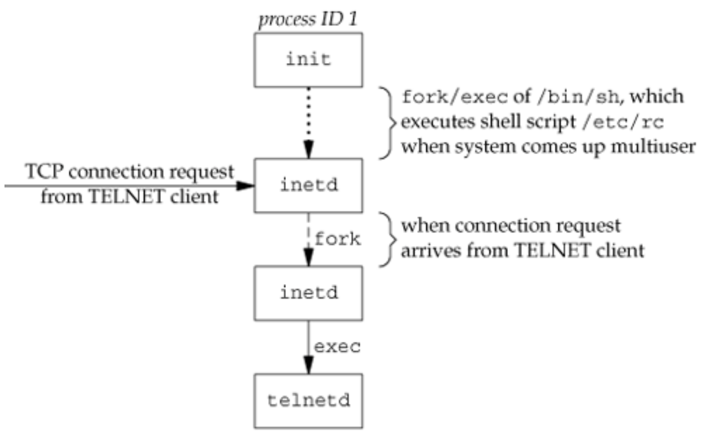
The main (physical) difference between logging in to a system through a serial terminal and logging in to a system through a network is that the connection between the terminal and the computer isn't point-to-point. With the terminal logins that we described in the previous section, `init` knows which terminal devices are enabled for logins and spawns a `getty` process for each device. In the case of network logins, however, all the logins come through the kernel's network interface drivers (e.g., the Ethernet driver).

Let's assume that a TCP connection request arrives for the TELNET server. TELNET is a remote login application that uses the TCP protocol. A user on another host (that is connected to the server's host through a network of some form) or on the same host initiates the login by starting the TELNET client:

```
    telnet hostname
```

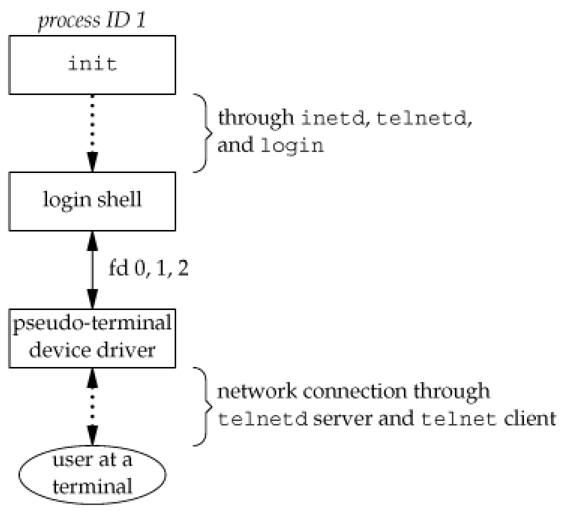
The client opens a TCP connection to hostname, and the program that's started on hostname is called the TELNET server. The client and the server then exchange data across the TCP connection using the TELNET application protocol. What has happened is that the user who started the client program is now logged in to the server's host. (This assumes, of course, that the user has a valid account on the server's host.) Figure 9.4 shows the sequence of processes involved in executing the TELNET server, called telnetd.

Figure 9.4. Sequence of processes involved in executing TELNET server


process ID 1
init
fork/exec of /bin/sh, which executes shell script /etc/rc when system comes up multiuser
TCP connection request from TELNET client
inetd
fork
when connection request arrives from TELNET client
inetd
exec
telnetd

The `telnetd` process then opens a pseudo-terminal device and splits into two processes using `fork`. The parent handles the communication across the network connection, and the child does an `exec` of the `login` program. The parent and the child are connected through the pseudo terminal. Before doing the `exec`, the child sets up file descriptors 0, 1, and 2 to the pseudo terminal. If we log in correctly, `login` performs the same steps we described in Section 9.2: it changes to our home directory and sets our group IDs, user ID, and our initial environment. Then `login` replaces itself with our login shell by calling `exec`. Figure 9.5 shows the arrangement of the processes at this point.

Figure 9.5. Arrangement of processes after everything is set for a network login


process ID 1
init
through inetd, telnetd, and login
login shell
fd 0, 1, 2
pseudo-terminal device driver
network connection through telnetd server and telnet client
user at a terminal

PROCESS GROUPS

A process group is a collection of one or more processes, usually associated with the same job, that can receive signals from the same terminal. Each process group has a unique process group ID. Process group IDs are similar to process IDs: they are positive integers and can be stored in a `pid_t` data type. The function `getpgrp` returns the process group ID of the calling process.

```
#include <unistd.h>

pid_t getpgrp(void);
```

Returns: process group ID of calling process

The Single UNIX Specification defines the `getpgid` function as an XSI extension that mimics this behavior.

```
#include <unistd.h>

pid_t getpgid(pid_t pid);
```

Returns: process group ID if OK, 1 on error

Each process group can have a process group leader. The leader is identified by its process group ID being equal to its process ID.

It is possible for a process group leader to create a process group, create processes in the group, and then terminate. The process group still exists, as long as at least one process is in the group, regardless of whether the group leader terminates. This is called the process group lifetime-the period of time that begins when the group is created and ends when the last remaining process leaves the group. The last remaining process in the process group can either terminate or enter some other process group.

A process joins an existing process group or creates a new process group by calling `setpgid`.

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

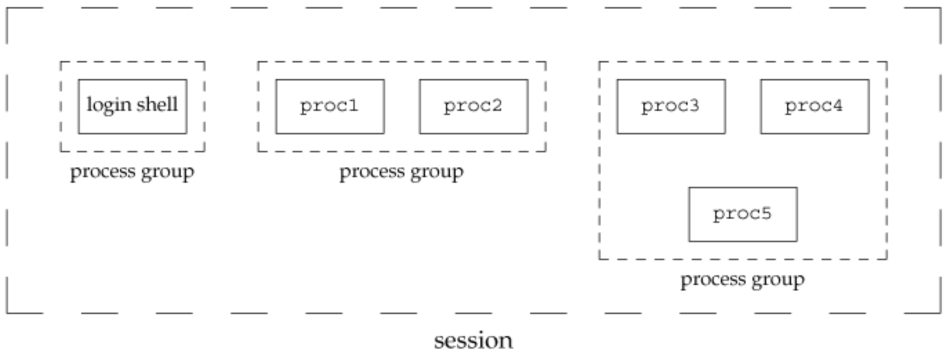
Returns: 0 if OK, 1 on error

This function sets the process group ID to pgid in the process whose process ID equals pid. If the two arguments are equal, the process specified by pid becomes a process group leader. If pid is 0, the process ID of the caller is used.

SESSIONS

A session is a collection of one or more process groups. For example, we could have the arrangement shown in Figure 9.6. Here we have three process groups in a single session.

Figure 9.6. Arrangement of processes into process groups and sessions



A process establishes a new session by calling the `setsid` function.

```
#include <unistd.h>

pid_t setsid(void);
```

Returns: process group ID if OK, 1 on error

If the calling process is not a process group leader, this function creates a new session. Three things happen.

- The process becomes the session leader of this new session. (A session leader is the process that creates a session.) The process is the only process in this new session.
- The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.
- The process has no controlling terminal. If the process had a controlling terminal before calling `setsid`, that association is broken.

This function returns an error if the caller is already a process group leader. The `getsid` function returns the process group ID of a process's session leader. The `getsid` function is included as an XSI extension in the Single UNIX Specification.

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

Returns: session leader's process group ID if OK, 1 on error

If pid is 0, `getsid` returns the process group ID of the calling process's session leader.

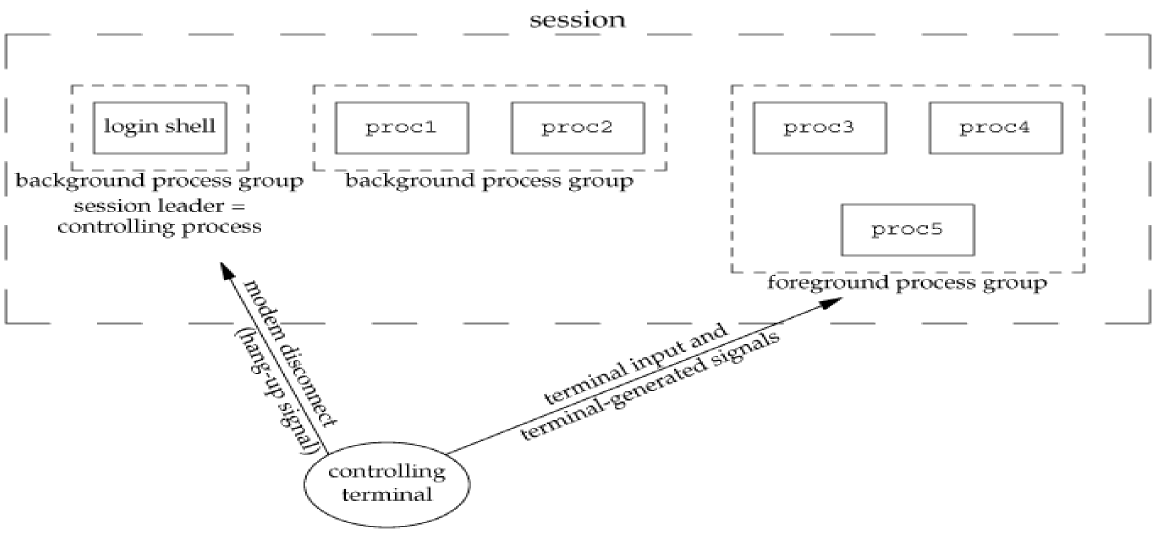
CONTROLLING TERMINAL

Sessions and process groups have a few other characteristics.

- A session can have a single controlling terminal. This is usually the terminal device (in the case of a terminal login) or pseudo-terminal device (in the case of a network login) on which we log in.
- The session leader that establishes the connection to the controlling terminal is called the controlling process.
- The process groups within a session can be divided into a single foreground process group and one or more background process groups.
- If a session has a controlling terminal, it has a single foreground process group, and all other process groups in the session are background process groups.
- Whenever we type the terminal's interrupt key (often DELETE or Control-C), this causes the interrupt signal be sent to all processes in the foreground process group.
- Whenever we type the terminal's quit key (often Control-backslash), this causes the quit signal to be sent to all processes in the foreground process group.
- If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

These characteristics are shown in Figure 9.7.

Figure 9.7. Process groups and sessions showing controlling terminal



tcgetpgrp, tcsetpgrp, AND tcgetsid FUNCTIONS

We need a way to tell the kernel which process group is the foreground process group, so that the terminal device driver knows where to send the terminal input and the terminal-generated signals.
To retrieve the foreground process group-id and to set the foreground process group-id we can use tcgetprgp and tcsetpgrp function.

The prototype of these functions are :

```
#include <unistd.h>
pid_t tcgetpgrp(int filedes);
```

Returns : process group ID of foreground process group if OK, -1 on error

```
int tcsetpgrp(int filedes, pid_t pgrpid);
```

Returns: 0 if OK, -1 on error

The function tcgetpgrp returns the process group ID of the foreground process group associated with the terminal open on *filedes.* If the process has a controlling terminal, the process can call tcsetpgrp to set the foreground process group ID to pgrpid. The value of pgrpid must be the process group ID of a process group in the same session, and filedes must refer to the controlling terminal of the session.
The single UNIX specification defines an XSI extension called tcgetsid to allow an application to obtain the process group-ID for the session leader given a file descriptor for the controlling terminal.

```
#include <termios.h>
pid_t tcgetsid(int filedes);
```

Returns: session leader's process group ID if Ok, -1 on error

JOB CONTROL

This feature allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are to run in the background. Job control requires three forms of support:

- A shell that supports job control
- The terminal driver in the kernel must support job control
- The kernel must support certain job-control signals

The interaction with the terminal driver arises because a special terminal character affects the foreground job: the suspend key (typically Control-Z). Entering this character causes the terminal driver to send the SIGTSTP signal to all processes in the foreground process group. The jobs in any background process groups aren't affected. The terminal driver looks for three special characters, which generate signals to the foreground process group.

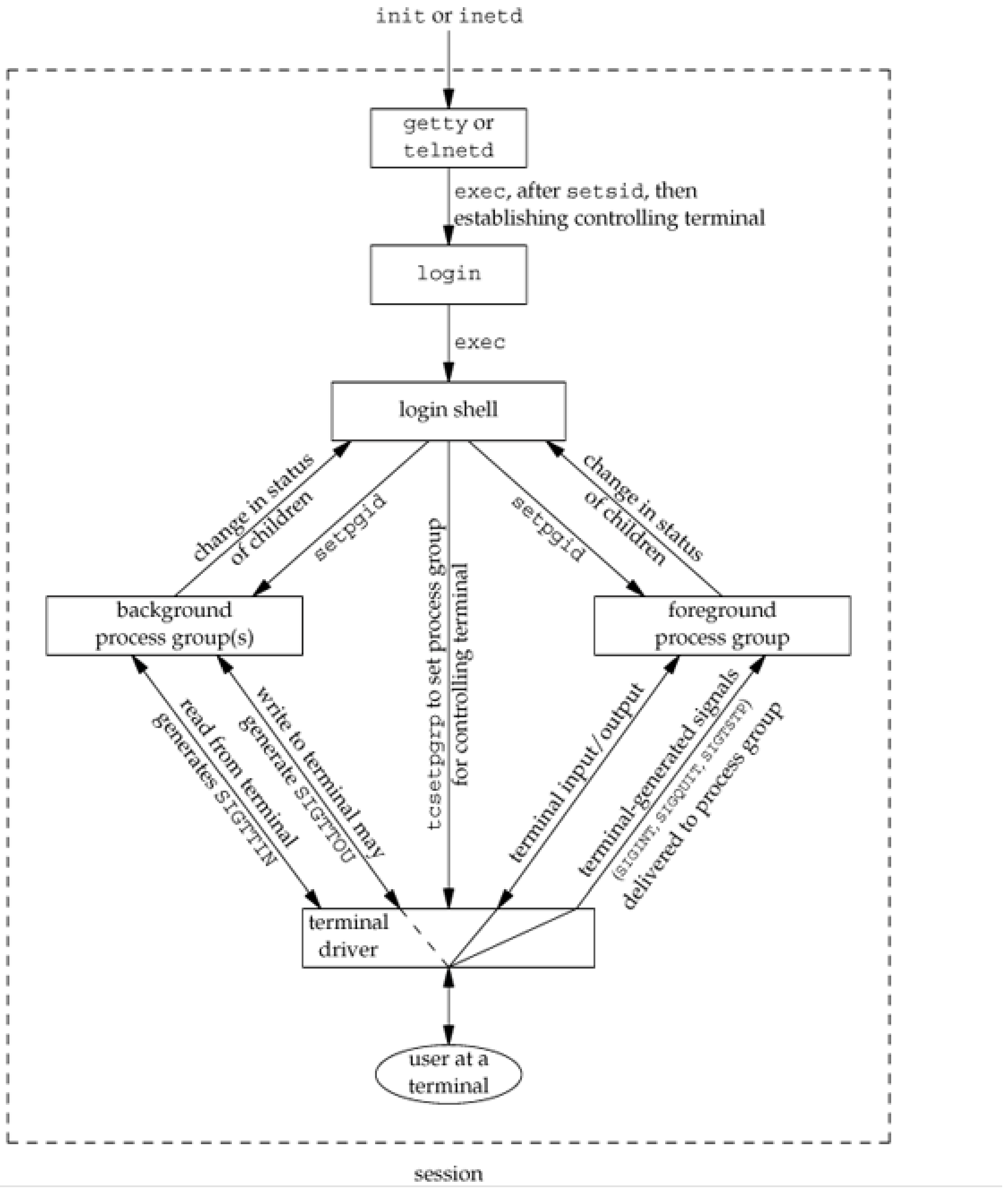
- The interrupt character (typically DELETE or Control-C) generates SIGINT.
- The quit character (typically Control-backslash) generates SIGQUIT.

- The suspend character (typically Control-Z) generates SIGTSTP.

This signal normally stops the background job; by using the shell, we are notified of this and can bring the job into the foreground so that it can read from the terminal. The following demonstrates this:

```
$ cat > temp.foo &          start in background, but it'll read from standard input
   [1]     1681
   $                           we press RETURN
   [1] + Stopped (SIGTTIN)     cat > temp.foo &
   $ fg %1                     bring job number 1 into the foreground
   cat > temp.foo              the shell tells us which job is now in the foreground

   hello, world                enter one line

   ^D                          type the end-of-file character
   $ cat temp.foo              check that the one line was put into the file
   hello, world
```

Figure 9.8. Summary of job control features with foreground & background jobs & terminal driver


init or inetd
getty or telnetd
exec, after setsid, then establishing controlling terminal
login
exec
login shell
change in status of children
setpgid
tcsetpgrp to set process group for controlling terminal
setpgid
change in status of children
background process group(s)
foreground process group
read from terminal generates SIGTTIN
write to terminal may generate SIGTTOU
terminal input/output
terminal-generated signals (SIGINT, SIGQUIT, SIGTSTP) delivered to process group
terminal driver
user at a terminal
session

What happens if a background job outputs to the controlling terminal? This is an option that we can allow or disallow. Normally, we use the `stty`(1) command to change this option. The following shows how this works:

```
  $ cat temp.foo &                      execute in background
  [1]     1719
  $ hello, world              the output from the background job appears after the prompt
                               we press RETURN

  [1] + Done      cat temp.foo &

  $ stty tostop           disable ability of background jobs to output to controlling terminal
  $ cat temp.foo &              try it again in the background
  [1]     1721
  $                             we press RETURN and find the job is stopped

  [1] + Stopped(SIGTTOU)           cat temp.foo &

  $ fg %1                       resume stopped job in the foreground
  cat temp.foo                  the shell tells us which job is now in the foreground
  hello, world                  and here is its output
```

When we disallow background jobs from writing to the controlling terminal, `cat` will block when it tries to write to its standard output, because the terminal driver identifies the write as coming from a background process and sends the job the `SIGTTOU` signal.
Figure 9.8 summarizes some of the features of job control that we've been describing. The solid lines through the terminal driver box mean that the terminal I/O and the terminal-generated signals are always connected from the foreground process group to the actual terminal. The dashed line corresponding to the `SIGTTOU` signal means that whether the output from a process in the background process group appears on the terminal is an option.

SHELL EXECUTION OF PROGRAMS

Example 1: **ps -o pid,ppid,pgid,sid,comm**
Output 1:

```
  PID  PPID  PGID  SID  COMMAND
  949   947   949  949  sh
 1774   949   949  949  ps
```

Example 2: **ps -o pid,ppid,pgid,sid,comm &**
Output 2:

```
  PID  PPID  PGID  SID COMMAND
  949   947   949  949 sh
 1812   949   949  949 ps
```

If we execute the command in the background, the only value that changes is the process ID of the command.

Example 3: **ps -o pid,ppid,pgid,sid,comm | cat1**
Output 3:

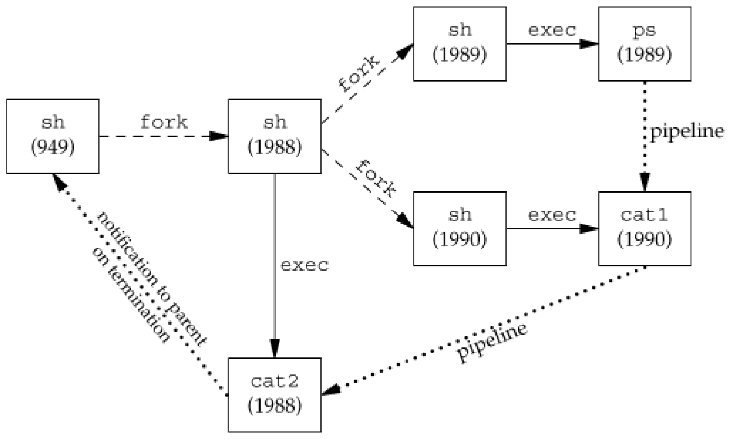
```
  PID  PPID  PGID  SID COMMAND
  949   947   949  949 sh
 1823   949   949  949 cat1
 1824  1823   949  949 ps
```

The program `cat1` is just a copy of the standard `cat` program, with a different name. Note that the last process in the pipeline is the child of the shell and that the first process in the pipeline is a child of the last process.

Example 4: **ps -o pid,ppid,pgid,sid,comm | cat1 | cat2**
Output 4:

```
  PID  PPID  PGID  SID COMMAND
  949   947   949  949 sh
 1988   949   949  949 cat2
 1989  1988   949  949 ps
 1990  1988   949  949 cat1
```

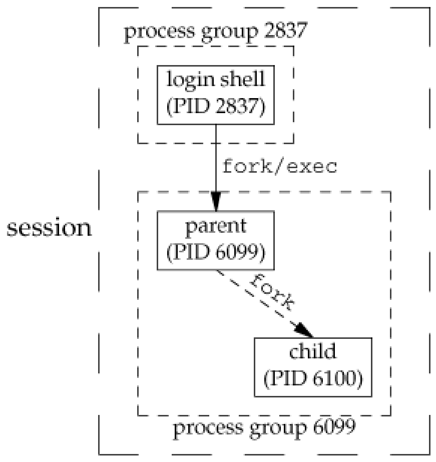
Figure 9.9. Processes in the pipeline `ps | cat1 | cat2` when invoked by Bourne shell



ORPHANED PROCESS GROUPS

A process whose parent terminates is called an orphan and is inherited by the `init` process.

Example of a process group about to be orphaned



Creating an orphaned process group

```
#include "apue.h"
#include <errno.h>

static void
sig_hup(int signo)
{
    printf("SIGHUP received, pid = %d\n", getpid());
}

static void
pr_ids(char *name)
{
    printf("%s: pid = %d, ppid = %d, pgrp = %d, tpgrp = %d\n",
        name, getpid(), getppid(), getpgrp(), tcgetpgrp(STDIN_FILENO));
    fflush(stdout);
}

int
main(void)
{
     char     c;
     pid_t    pid;

     pr_ids("parent");
     if ((pid = fork()) < 0) {
         err_sys("fork error");
     } else if (pid > 0) {   /* parent */
```

```
        sleep(5);       /*sleep to let child stop itself */
        exit(0);        /* then parent exits */
    } else {            /* child */
        pr_ids("child");
        signal(SIGHUP, sig_hup);    /* establish signal handler */
        kill(getpid(), SIGTSTP);    /* stop ourself */
        pr_ids("child");    /* prints only if we're continued */
        if (read(STDIN_FILENO, &c, 1) != 1)
            printf("read error from controlling TTY, errno = %d\n",
                errno);
        exit(0);
    }
}
```

Output:

```
 $ ./a.out
parent: pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099
child: pid = 6100, ppid = 6099, pgrp = 6099, tpgrp = 6099
$ SIGHUP received, pid = 6100
child: pid = 6100, ppid = 1, pgrp = 6099, tpgrp = 2837
read error from controlling TTY, errno = 5
```

The child inherits the process group of its parent (6099). After the `fork`,

- The parent sleeps for 5 seconds. This is our (imperfect) way of letting the child execute before the parent terminates.
- The child establishes a signal handler for the hang-up signal (`SIGHUP`). This is so we can see whether `SIGHUP` is sent to the child. The child sends itself the stop signal (`SIGTSTP`) with the `kill` function. This stops the child, similar to our stopping a foreground job with our terminal's suspend character (Control-Z).
- When the parent terminates, the child is orphaned, so the child's parent process ID becomes 1, the `init` process ID.
- At this point, the child is now a member of an orphaned process group If the process group is not orphaned, there is a chance that one of those parents in a different process group but in the same session will restart a stopped process in the process group that is not orphaned. Here, the parent of every process in the group belongs to another session.
- Since the process group is orphaned when the parent terminates, POSIX.1 requires that every process in the newly orphaned process group that is stopped (as our child is) be sent the hang-up signal (`SIGHUP`) followed by the continue signal (`SIGCONT`).
- This causes the child to be continued, after processing the hang-up signal. The default action for the hang-up signal is to terminate the process, so we have to provide a signal handler to catch the signal. We therefore expect the `printf` in the `sig_hup` function to appear before the `printf` in the `pr_ids` function.

UNIT 6
SIGNALS AND DAEMON PROCESSES

Signals are software interrupts. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

Name	Description	Default action
SIGABRT	abnormal termination (`abort`)	terminate+core
SIGALRM	timer expired (`alarm`)	terminate
SIGBUS	hardware fault	terminate+core
SIGCANCEL	threads library internal use	ignore
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGEMT	hardware fault	terminate+core
SIGFPE	arithmetic exception	terminate+core
SIGFREEZE	checkpoint freeze	ignore
SIGHUP	hangup	terminate
SIGILL	illegal instruction	terminate+core
SIGINFO	status request from keyboard	ignore
SIGINT	terminal interrupt character	terminate
SIGIO	asynchronous I/O	terminate/ignore
SIGIOT	hardware fault	terminate+core
SIGKILL	termination	terminate
SIGLWP	threads library internal use	ignore
SIGPIPE	write to pipe with no readers	terminate
SIGPOLL	pollable event (`poll`)	terminate
SIGPROF	profiling time alarm (`setitimer`)	terminate
SIGPWR	power fail/restart	terminate/ignore
SIGQUIT	terminal quit character	terminate+core
SIGSEGV	invalid memory reference	terminate+core
SIGSTKFLT	coprocessor stack fault	terminate
SIGSTOP	stop	stop process
SIGSYS	invalid system call	terminate+core
SIGTERM	termination	terminate
SIGTHAW	checkpoint thaw	ignore
SIGTRAP	hardware fault	terminate+core
SIGTSTP	terminal stop character	stop process
SIGTTIN	background read from control tty	stop process

SIGTTOU	background write to control tty	stop process
SIGURG	urgent condition (sockets)	ignore
SIGUSR1	user-defined signal	terminate
SIGUSR2	user-defined signal	terminate
SIGVTALRM	virtual time alarm (setitimer)	terminate
SIGWAITING	threads library internal use	ignore
SIGWINCH	terminal window size change	ignore
SIGXCPU	CPU limit exceeded (setrlimit)	terminate+core/ignore
SIGXFSZ	file size limit exceeded (setrlimit)	terminate+core/ignore
SIGXRES	resource control exceeded	Ignore

When a signal is sent to a process, it is pending on the process to handle it. The process can react to pending signals in one of three ways:

- Accept the **default action** of the signal, which for most signals will terminate the process.
- **Ignore** the signal. The signal will be discarded and it has no affect whatsoever on the recipient process.
- Invoke a **user-defined** function. The function is known as a signal handler routine and the signal is said to be ***caught*** when this function is called.

THE UNIX KERNEL SUPPORT OF SIGNALS

- When a signal is generated for a process, the kernel will set the corresponding signal flag in the process table slot of the recipient process.
- If the recipient process is asleep, the kernel will awaken the process by scheduling it.
- When the recipient process runs, the kernel will check the process U-area that contains an array of signal handling specifications.
- If array entry contains a zero value, the process will accept the default action of the signal.
- If array entry contains a 1 value, the process will ignore the signal and kernel will discard it.
- If array entry contains any other value, it is used as the function pointer for a user-defined signal handler routine.

SIGNAL

The function prototype of the signal API is:

```
#include <signal.h>


void (*signal(int sig_no, void (*handler)(int)))(int);
```

The formal argument of the API are: sig_no is a signal identifier like SIGINT or SIGTERM. The handler argument is the function pointer of a user-defined signal handler function.

The following example attempts to catch the SIGTERM signal, ignores the SIGINT signal, and accepts the default action of the SIGSEGV signal. The pause API suspends the calling process until it is interrupted by a signal and the corresponding signal handler does a return:

```
#include<iostream.h>
#include<signal.h>
/*signal handler function*/
void catch_sig(int sig_num)
{
      signal (sig_num,catch_sig);
      cout<<"catch_sig:"<<sig_num<<endl;
}

/*main function*/
int main()
{
      signal(SIGTERM,catch_sig);
      signal(SIGINT,SIG_IGN);
      signal(SIGSEGV,SIG_DFL);
      pause( );               /*wait for a signal interruption*/
}
```

The SIG_IGN specifies a signal is to be ignored, which means that if the signal is generated to the process, it will be discarded without any interruption of the process.

The SIG_DFL specifies to accept the default action of a signal.

SIGNAL MASK

A process initially inherits the parent's signal mask when it is created, but any pending signals for the parent process are not passed on. A process may query or set its signal mask via the sigprocmask API:

```
#include <signal.h>
int sigprocmask(int cmd, const sigset_t *new_mask, sigset_t *old_mask);
```

Returns: 0 if OK, 1 on error

The new_mask argument defines a set of signals to be set or reset in a calling process signal mask, and the cmd argument specifies how the new_mask value is to be used by the API. The possible values of cmd and the corresponding use of the new_mask value are:

Cmd value	Meaning
SIG_SETMASK	Overrides the calling process signal mask with the value specified in the new_mask argument.
SIG_BLOCK	Adds the signals specified in the new_mask argument to the calling process signal mask.
SIG_UNBLOCK	Removes the signals specified in the new_mask argument from the calling process signal mask.

- ✓ If the actual argument to new_mask argument is a NULL pointer, the cmd argument will be ignored, and the current process signal mask will not be altered.
- ✓ If the actual argument to old_mask is a NULL pointer, no previous signal mask will be returned.
- ✓ The sigset_t contains a collection of bit flags.

The BSD UNIX and POSIX.1 define a set of API known as sigsetops functions:

```
#include<signal.h>

int sigemptyset (sigset_t* sigmask);
int sigaddset (sigset_t* sigmask, const int sig_num);
int sigdelset (sigset_t* sigmask, const int sig_num);
int sigfillset (sigset_t* sigmask);
int sigismember (const sigset_t* sigmask, const int sig_num);
```

- The sigemptyset API clears all signal flags in the sigmask argument.
- The sigaddset API sets the flag corresponding to the signal_num signal in the sigmask argument.
- The sigdelset API clears the flag corresponding to the signal_num signal in the sigmask argument.
- The sigfillset API sets all the signal flags in the sigmask argument.
  [all the above functions return 0 if OK, -1 on error]
- The sigismember API returns 1 if flag is set, 0 if not set and -1 if the call fails.

The following example checks whether the SIGINT signal is present in a process signal mask and adds it to the mask if it is not there.

```
#include<stdio.h>
#include<signal.h>
int main()
{
        sigset_t     sigmask;
        sigemptyset(&sigmask);           /*initialise set*/

        if(sigprocmask(0,0,&sigmask)==-1)       /*get current signal mask*/
        {
                perror("sigprocmask");
                exit(1);
        }
        else sigaddset(&sigmask,SIGINT);  /*set SIGINT flag*/

        sigdelset(&sigmask, SIGSEGV);           /*clear SIGSEGV flag*/
        if(sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
                perror("sigprocmask");
}
```

A process can query which signals are pending for it via the sigpending API:

```
#include<signal.h>
int sigpending(sigset_t* sigmask);
```

Returns 0 if OK, -1 if fails.

The sigpending API can be useful to find out whether one or more signals are pending for a process and to set up special signal handling methods for these signals before the process calls the sigprocmask API to unblock them.

The following example reports to the console whether the SIGTERM signal is pending for the process:

```
#include<iostream.h>
#include<stdio.h>
#include<signal.h>
int main()
{
        sigset_t    sigmask;
        sigemptyset(&sigmask);
        if(sigpending(&sigmask)==-1)
                perror("sigpending");
```

```
      else cout << "SIGTERM signal is:"
            << (sigismember(&sigmask,SIGTERM) ? "Set" : "No Set")  << endl;
}
```

In addition to the above, UNIX also supports following APIs for signal mask manipulation:

```
#include<signal.h>

int sighold(int signal_num);
int sigrelse(int signal_num);
int sigignore(int signal_num);
int sigpause(int signal_num);
```

SIGACTION

The sigaction API blocks the signal it is catching allowing a process to specify additional signals to be blocked when the API is handling a signal.

The sigaction API prototype is:

```
#include<signal.h>
int sigaction(int signal_num, struct sigaction* action, struct sigaction* old_action);
```

Returns: 0 if OK, 1 on error

The struct sigaction data type is defined in the <signal.h> header as:

```
struct sigaction
{
      void          (*sa_handler)(int);
      sigset_t      sa_mask;
      int           sa_flag;
}
```

The following program illustrates the uses of sigaction:

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

void callme(int sig_num)
{
      cout<<"catch signal:"<<sig_num<<endl;
}
int main(int argc, char* argv[])
{
      sigset_t sigmask;
      struct sigaction action,old_action;
      sigemptyset(&sigmask);
      if(sigaddset(&sigmask,SIGTERM)==-1 || sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
            perror("set signal mask");
```

```
        sigemptyset(&action.sa_mask);
        sigaddset(&action.sa_mask,SIGSEGV);
        action.sa_handler=callme;
        action.sa_flags=0;
        if(sigaction(SIGINT,&action,&old_action)==-1)
                perror("sigaction");
        pause();
        cout<<argv[0]<<"exists\n";
        return 0;
}
```

THE SIGCHLD SIGNAL AND THE waitpid API

When a child process terminates or stops, the kernel will generate a SIGCHLD signal to its parent process. Depending on how the parent sets up the handling of the SIGCHLD signal, different events may occur:

- Parent accepts the **default action** of the SIGCHLD signal:
  - SIGCHLD does not terminate the parent process.
  - Parent process will be awakened.
  - API will return the child's exit status and process ID to the parent.
  - Kernel will clear up the Process Table slot allocated for the child process.
  - Parent process can call the waitpid API repeatedly to wait for each child it created.
- Parent **ignores** the SIGCHLD signal:
  - SIGCHLD signal will be discarded.
  - Parent will not be disturbed even if it is executing the waitpid system call.
  - If the parent calls the waitpid API, the API will suspend the parent until all its child processes have terminated.
  - Child process table slots will be cleared up by the kernel.
  - API will return a -1 value to the parent process.
- Process **catches** the SIGCHLD signal:
  - The signal handler function will be called in the parent process whenever a child process terminates.
  - If the SIGCHLD arrives while the parent process is executing the waitpid system call, the waitpid API may be restarted to collect the child exit status and clear its process table slots.
  - Depending on parent setup, the API may be aborted and child process table slot not freed.

THE sigsetjmp AND siglongjmp APIs

The function prototypes of the APIs are:

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
int siglongjmp(sigjmp_buf env, int val);
```

The sigsetjmp and siglongjmp are created to support signal mask processing. Specifically, it is implementation-dependent on whether a process signal mask is saved and restored when it invokes the setjmp and longjmp APIs respectively.

The only difference between these functions and the setjmp and longjmp functions is that sigsetjmp has an additional argument. If savemask is nonzero, then sigsetjmp also saves the current signal mask of the process in env. When siglongjmp is called, if the env argument was saved by a call to sigsetjmp with a nonzero savemask, then siglongjmp restores the saved signal mask. The siglongjmp API is usually called from user-defined signal handling functions. This is because a process signal mask is modified when a signal handler is called, and siglongjmp should be called to ensure the process signal mask is restored properly when "jumping out" from a signal handling function.

The following program illustrates the uses of sigsetjmp and siglongjmp APIs.

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

sigjmp_buf   env;
void callme(int sig_num)
{
       cout<< "catch signal:" <<sig_num <<endl;
       siglongjmp(env,2);
}
int main()
{
       sigset_t      sigmask;
       struct sigaction action,old_action;

       sigemptyset(&sigmask);

       if(sigaddset(&sigmask,SIGTERM)==-1) || sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
              perror("set signal mask");
       sigemptyset(&action.sa_mask);
       sigaddset(&action.sa_mask,SIGSEGV);
       action.sa_handler=(void(*)())callme;
       action.sa_flags=0;
       if(sigaction(SIGINT,&action,&old_action)==-1)
              perror("sigaction");
       if(sigsetjmp(env,1)!=0)
       {
              cerr<<"return from signal interruption";
              return 0;
       }
       else
              cerr<<"return from first time sigsetjmp is called";
       pause();
}
```

KILL

A process can send a signal to a related process via the kill API. This is a simple means of inter-process communication or control. The function prototype of the API is:

```
#include<signal.h>
int kill(pid_t pid, int signal_num);
```

Returns: 0 on success, -1 on failure.

The signal_num argument is the integer value of a signal to be sent to one or more processes designated by pid. The possible values of pid and its use by the kill API are:

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 the signal.
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	The signal is sent to all processes on the system for which the sender has permission to send the signal.

The following program illustrates the implementation of the UNIX kill command using the kill API:

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

int main(int argc,char** argv)
{
        int pid, sig = SIGTERM;
        if(argc==3)
        {
                if(sscanf(argv[1],"%d",&sig)!=1)
                {
                        cerr<<"invalid number:" << argv[1] << endl;
                        return -1;
                }
                argv++,argc--;
        }
        while(--argc>0)
        if(sscanf(*++argv, "%d", &pid)==1)
        {
                if(kill(pid,sig)==-1)
                        perror("kill");
        }
        else
                cerr<<"invalid pid:" << argv[0] <<endl;
        return 0;
}
```

The UNIX kill command invocation syntax is:

Kill [-<signal_num>] <pid>......

Where signal_num can be an integer number or the symbolic name of a signal. <pid> is process ID.

ALARM

The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds. The function prototype of the API is:

```
#include<signal.h>
Unsigned int alarm(unsigned int time_interval);
```

Returns: 0 or number of seconds until previously set alarm

The alarm API can be used to implement the sleep API:

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

void wakeup( )
{ ; }

unsigned int sleep (unsigned int timer )
{
       struct sigaction action;
       action.sa_handler=wakeup;
       action.sa_flags=0;
       sigemptyset(&action.sa_mask);
       if(sigaction(SIGALARM,&action,0)==-1)
       {
              perror("sigaction");
              return -1;
       }
       (void) alarm (timer);
       (void) pause( );
       return 0;
 }
```

INTERVAL TIMERS

The interval timer can be used to schedule a process to do some tasks at a fixed time interval, to time the execution of some operations, or to limit the time allowed for the execution of some tasks.

The following program illustrates how to set up a real-time clock interval timer using the alarm API:

```
 #include<stdio.h>
 #include<unistd.h>
 #include<signal.h>
 #define INTERVAL 5
void callme(int sig_no)
{
       alarm(INTERVAL);
        /*do scheduled tasks*/
}
int main()
```

```
{
        struct sigaction action;
        sigemptyset(&action.sa_mask);
        action.sa_handler=(void(*)( )) callme;
        action.sa_flags=SA_RESTART;
        if(sigaction(SIGALARM,&action,0)==-1)
        {
                perror("sigaction");
                return 1;
        }
        if(alarm(INTERVAL)==-1)
                 perror("alarm");
        else while(1)
        {
                /*do normal operation*/
        }
        return 0;
}
```

In addition to alarm API, UNIX also invented the setitimer API, which can be used to define up to three different types of timers in a process:

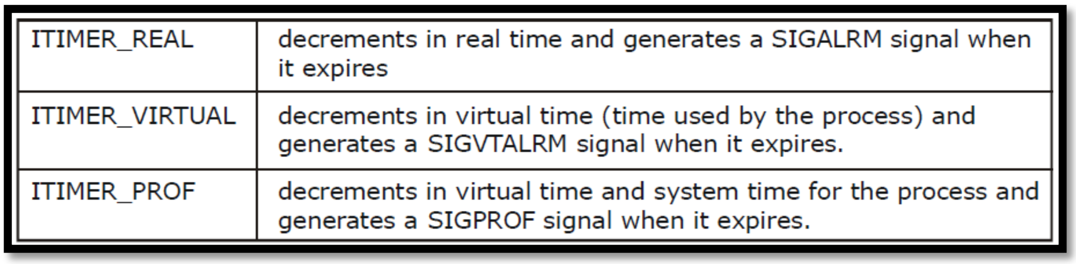
- Real time clock timer
- Timer based on the user time spent by a process
- Timer based on the total user and system times spent by a process

The getitimer API is also defined for users to query the timer values that are set by the setitimer API.

The setitimer and getitimer function prototypes are:

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

int setitimer(int which, const struct itimerval * val, struct itimerval * old);
int getitimer(int which, struct itimerval * old);
```

The *which* arguments to the above APIs specify which timer to process. Its possible values and the corresponding timer types are:



ITIMER_REAL	decrements in real time and generates a SIGALRM signal when it expires
ITIMER_VIRTUAL	decrements in virtual time (time used by the process) and generates a SIGVTALRM signal when it expires.
ITIMER_PROF	decrements in virtual time and system time for the process and generates a SIGPROF signal when it expires.

The struct itimerval datatype is defined as:

```
struct itimerval
{
        struct timeval it_value;          /*current value*/
        struct timeval it_interval;       /* time interval*/
};
```

Example program:

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#define INTERVAL 5
void callme(int sig_no)
{
        /*do scheduled tasks*/
}
int main()
{
       struct itimerval val;
       struct sigaction action;

       sigemptyset(&action.sa_mask);
       action.sa_handler=(void(*)( )) callme;
       action.sa_flags=SA_RESTART;
       if(sigaction(SIGALARM,&action,0)==-1)
       {
              perror("sigaction");
              return 1;
       }
       val.it_interval.tv_sec     =INTERVAL;
       val.it_interval.tv_usec    =0;
       val.it_value.tv_sec        =INTERVAL;
       val.it_value.tv_usec       =0;

       if(setitimer(ITIMER_REAL, &val , 0)==-1)
              perror("alarm");

       else while(1)
       {
              /*do normal operation*/
       }
       return 0;
}
```

The setitimer and getitimer APIs return a zero value if they succeed or a -1 value if they fail.

POSIX.1b TIMERS

POSIX.1b defines a set of APIs for interval timer manipulations. The POSIX.1b timers are more flexible and powerful than are the UNIX timers in the following ways:

- Users may define multiple independent timers per system clock.
- The timer resolution is in nanoseconds.
- Users may specify the signal to be raised when a timer expires.
- The time interval may be specified as either an absolute or a relative time.

The POSIX.1b APIs for timer manipulations are:

```
#include<signal.h>
#include<time.h>

int timer_create(clockid_t clock, struct sigevent* spec, timer_t* timer_hdrp);
int timer_settime(timer_t timer_hdr, int flag, struct itimerspec* val, struct itimerspec* old);
int timer_gettime(timer_t timer_hdr, struct itimerspec* old);
int timer_getoverrun(timer_t timer_hdr);
int timer_delete(timer_t timer_hdr);
```

DAEMON PROCESSES

INTRODUCTION

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down.

DAEMON CHARACTERISTICS

The characteristics of daemons are:

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don’t have controlling terminal.
- Daemons are session and group leaders.

CODING RULES

- **Call `umask` to set the file mode creation mask to 0**. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions.
- **Call `fork` and have the parent `exit`**. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader.
- **Call `setsid` to create a new session**. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.
- **Change the current working directory to the root directory**. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.
- **Unneeded file descriptors should be closed.** This prevents the daemon from holding open any descriptors that it may have inherited from its parent.
- **Some daemons open file descriptors 0, 1, and 2 to `/dev/null` so that any library routines that try to read from standard input or write to standard output or standard error will have no effect**. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed; nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

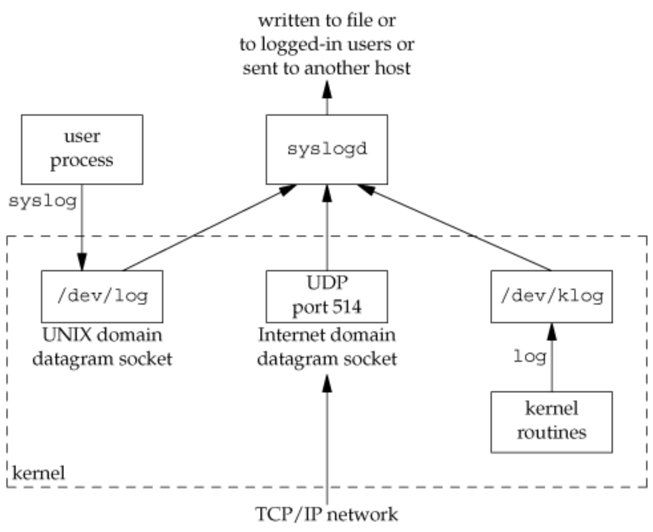
Example Program:

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

int daemon_initialise( )
 {
      pid_t pid;
      if (( pid = for() ) < 0)
             return -1;
      else if ( pid != 0)
             exit(0);       /* parent exits */

      /* child continues */
      setsid( );
      chdir("/");
      umask(0);
      return 0;
}
```

ERROR LOGGING

One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, since on many workstations, the console device runs a windowing system. A central daemon error-logging facility is required.

Figure 13.2. The BSD syslog facility



There are three ways to generate log messages:

- Kernel routines can call the `log` function. These messages can be read by any user process that `open`s and `read`s the `/dev/klog` device.
- Most user processes (daemons) call the `syslog`(3) function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket `/dev/log`.
- A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the `syslog` function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

Normally, the syslogd daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually /etc/syslog.conf, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file. Our interface to this facility is through the syslog function.

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
int setlogmask(int maskpri);
```

SINGLE-INSTANCE DAEMONS

Some daemons are implemented so that only a single copy of the daemon should be running at a time for proper operation. The file and record-locking mechanism provides the basis for one way to ensure that only one copy of a daemon is running. If each daemon creates a file and places a write lock on the entire file, only one such write lock will be allowed to be created. Successive attempts to create write locks will fail, serving as an indication to successive copies of the daemon that another instance is already running.

File and record locking provides a convenient mutual-exclusion mechanism. If the daemon obtains a write-lock on an entire file, the lock will be removed automatically if the daemon exits. This simplifies recovery, removing the need for us to clean up from the previous instance of the daemon.

PROGRAM:Ensure that only one copy of a daemon is running

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

extern int lockfile(int);

int already_running(void)
{
    int     fd;
    char    buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
        syslog(LOG_ERR, "can't open %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "can't lock %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}
```

DAEMON CONVENTIONS

- If the daemon uses a lock file, the file is usually stored in `/var/run`. Note, however, that the daemon might need superuser permissions to create a file here. The name of the file is usually `name.pid`, where name is the name of the daemon or the service. For example, the name of the `cron` daemon's lock file is `/var/run/crond.pid`.

- If the daemon supports configuration options, they are usually stored in `/etc`. The configuration file is named name`.conf`, where name is the name of the daemon or the name of the service. For example, the configuration for the `syslogd` daemon is `/etc/syslog.conf`.
- Daemons can be started from the command line, but they are usually started from one of the system initialization scripts (`/etc/rc*` or `/etc/init.d/*`). If the daemon should be restarted automatically when it exits, we can arrange for `init` to restart it if we include a `respawn` entry for it in `/etc/inittab`.
- If a daemon has a configuration file, the daemon reads it when it starts, but usually won't look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch `SIGHUP` and reread their configuration files when they receive the signal. Since they aren't associated with terminals and are either session leaders without controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive `SIGHUP`. Thus, they can safely reuse it

CLIENT-SERVER MODEL

In general, a server is a process that waits for a client to contact it, requesting some type of service. In Figure 13.2 [REFER PAGE 10], the service being provided by the `syslogd` server is the logging of an error message.

In Figure 13.2, the communication between the client and the server is one-way. The client sends its service request to the server; the server sends nothing back to the client. In the upcoming chapters, we'll see numerous examples of two-way communication between a client and a server. The client sends a request to the server, and the server sends a reply back to the client.

UNIT 7
INTERPROCESS COMMUNICATION

INTRODUCTION

IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems.
The various forms of IPC that are supported on a UNIX system are as follows :
1) Half duplex Pipes.
2) FIFO's
3) Full duplex Pipes.
4) Named full duplex Pipes.
5) Message queues.
6) Shared memory.
7) Semaphores.
8) Sockets.
9) STREAMS.
The first seven forms of IPC are usually restricted to IPC between processes on the same host. The final two i.e. Sockets and STREAMS are the only two that are generally supported for IPC between processes on different hosts.

PIPES

Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.

- Historically, they have been half duplex (i.e., data flows in only one direction).
- Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls `fork`, and the pipe is used between the parent and the child.

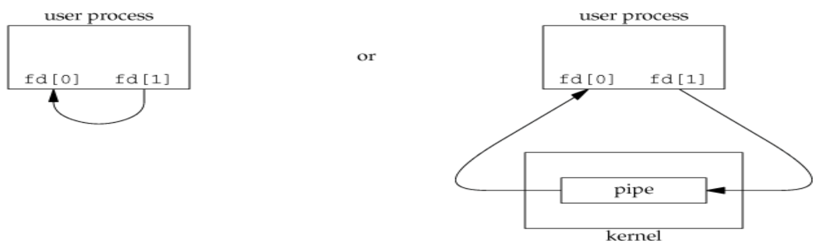
A pipe is created by calling the `pipe` function.

```
#include <unistd.h>

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

Returns: 0 if OK, 1 on error.
Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

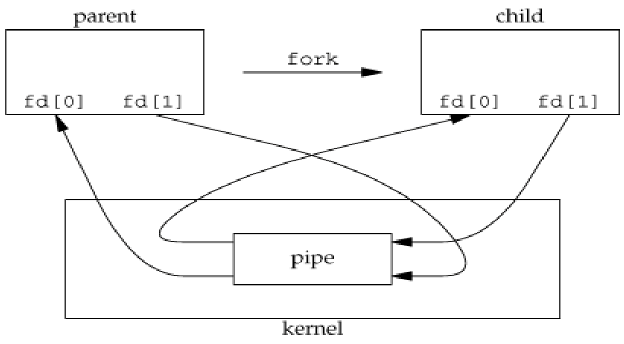
Two ways to picture a half-duplex pipe are shown in Figure 15.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

Figure 15.2. Two ways to view a half-duplex pipe



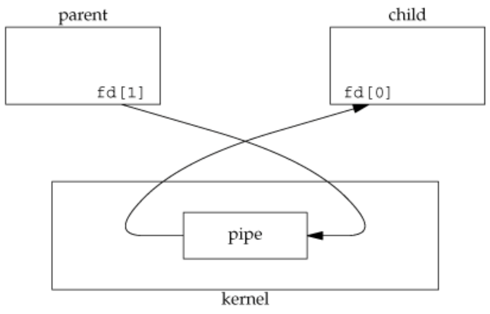
A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child or vice versa. Figure 15.3 shows this scenario.

Figure 15.3 Half-duplex pipe after a `fork`



What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]). Figure 15.4 shows the resulting arrangement of descriptors.

Figure 15.4. Pipe from parent to child



For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`.

When one end of a pipe is closed, the following two rules apply.

- ❖ If we `read` from a pipe whose write end has been closed, `read` returns 0 to indicate an end of file after all the data has been read.
- ❖ If we `write` to a pipe whose read end has been closed, the signal `SIGPIPE` is generated. If we either ignore the signal or catch it and return from the signal handler, `write` returns 1 with `errno` set to `EPIPE`.

PROGRAM: shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

popen AND pclose FUNCTIONS

Since a common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has historically provided the `popen` and `pclose` functions. These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, `fork`ing a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

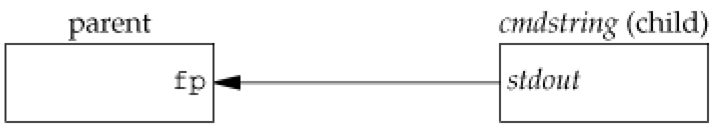
Returns: file pointer if OK, `NULL` on error

```
int pclose(FILE *fp);
```

Returns: termination status of cmdstring, or 1 on error

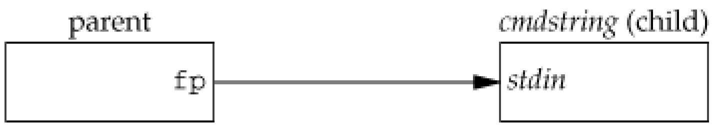
The function `popen` does a `fork` and `exec` to execute the cmdstring, and returns a standard I/O file pointer. If type is `"r"`, the file pointer is connected to the standard output of cmdstring

Figure 15.9. Result of `fp = popen`(cmdstring, `"r"`)



If type is `"w"`, the file pointer is connected to the standard input of cmdstring, as shown:

Figure 15.10. Result of `fp = popen`(cmdstring, `"w"`)

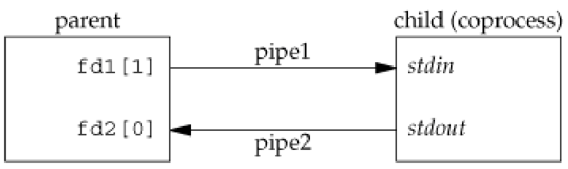


COPROCESSES

A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a coprocess when the same program generates the filter's input and reads the filter's output. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe.

The process creates two pipes: one is the standard input of the coprocess, and the other is the standard output of the coprocess. Figure 15.16 shows this arrangement.

Figure 15.16. Driving a coprocess by writing its standard input and reading its standard output



Program: Simple filter to add two numbers

```
#include "apue.h"

Int main(void)
{
    int     n,  int1,  int2;
    char    line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;        /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

FIFOs

FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe.

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

int mkfifo(const char *pathname, mode_t mode);
```

Returns: 0 if OK, 1 on error

Once we have used mkfifo to create a FIFO, we open it using open. When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.

- In the normal case (O_NONBLOCK not specified), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading.
- If O_NONBLOCK is specified, an open for read-only returns immediately. But an open for write-only returns 1 with errno set to ENXIO if no process has the FIFO open for reading.

There are two uses for FIFOs.

- ✓ FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
- ✓ FIFOs are used as rendezvous points in client-server applications to pass data between the clients and the servers.

Example Using FIFOs to Duplicate Output Streams

FIFOs can be used to duplicate an output stream in a series of shell commands. This prevents writing the data to an intermediate disk file. Consider a procedure that needs to process a filtered input stream twice. Figure 15.20 shows this arrangement.

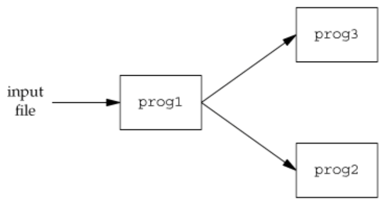


FIGURE 15.20 : Procedure that processes a filtered input stream twice

With a FIFO and the UNIX program tee(1), we can accomplish this procedure without using a temporary file. (The tee program copies its standard input to both its standard output and to the file named on its command line.)

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

We create the FIFO and then start prog3 in the background, reading from the FIFO. We then start prog1 and use tee to send its input to both the FIFO and prog2. Figure 15.21shows the process arrangement.

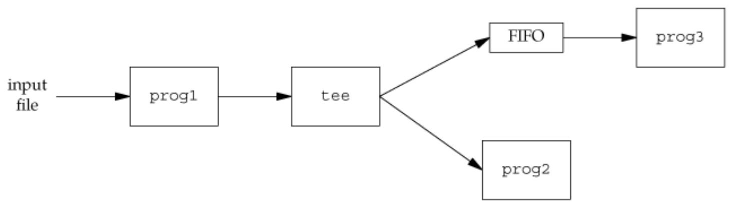


FIGURE 15.21 : Using a FIFO and `tee` to send a stream to two different processes

Example Client-Server Communication Using a FIFO

- FIFO's can be used to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
- This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server communication is how to send replies back from the server to each client.
- A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client'sprocess ID.
- For example, the server can create a FIFO with the name /vtu/ ser.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the file system.
- The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

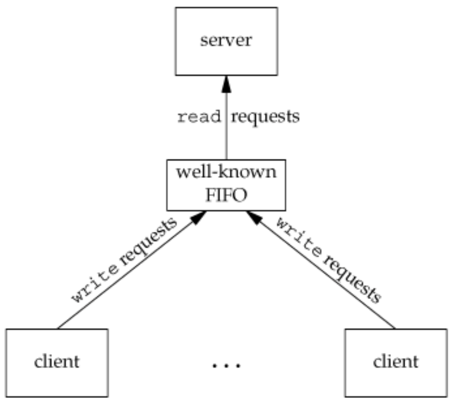


Figure 15.22. Clients sending requests to a server using a FIFO

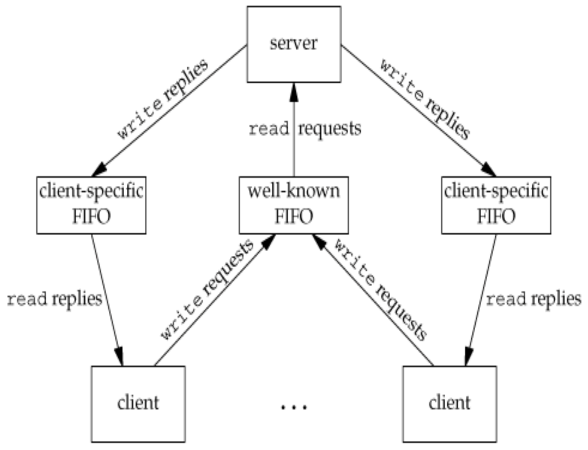


Figure 15.23. Client-server communication using FIFOs

XSI IPC

❖ Identifiers and Keys

Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer identifier. The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a key that acts as an external name.

Whenever an IPC structure is being created, a key must be specified. The data type of this key is the primitive system data type `key_t`, which is often defined as a long integer in the header `<sys/types.h>`. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

- The server can create a new IPC structure by specifying a key of `IPC_PRIVATE` and store the returned identifier somewhere (such as a file) for the client to obtain. The key `IPC_PRIVATE` guarantees that the server creates a new IPC structure. The disadvantage to this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.
  The `IPC_PRIVATE` key is also used in a parent-child relationship. The parent creates a new IPC structure specifying `IPC_PRIVATE`, and the resulting identifier is then available to the child after the `fork`. The child can pass the identifier to a new program as an argument to one of the `exec` functions.
- The client and the server can agree on a key by defining the key in a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the `get` function (`msgget`, `semget`, or `shmget`) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.
- The client and the server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function `ftok` to convert these two values into a key. This key is then used in step 2. The only service provided by `ftok` is a way of generating a key from a pathname and project ID.

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

key_t ftok(const char *path, int id);
```

Returns: key if OK, `(key_t)`-1 on error

The path argument must refer to an existing file. Only the lower 8 bits of id are used when generating the key.

The key created by `ftok` is usually formed by taking parts of the `st_dev` and `st_ino` fields in the `stat` structure corresponding to the given pathname and combining them with the project ID. If two pathnames refer to two different files, then `ftok` usually returns two different keys for the two pathnames. However, because both i-node numbers and keys are often stored in long integers, there can be information loss creating a key. This means that two different pathnames to different files can generate the same key if the same project ID is used.

❖ Permission Structure

XSI IPC associates an `ipc_perm` structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
struct ipc_perm
{
        uid_t  uid;  /* owner's effective user id */
        gid_t  gid;  /* owner's effective group id */
        uid_t  cuid; /* creator's effective user id */
        gid_t  cgid; /* creator's effective group id */
        mode_t mode; /* access modes */
        .
        .
        .
};
```

All the fields are initialized when the IPC structure is created. At a later time, we can modify the uid, gid, and mode fields by calling msgctl, semctl, or shmctl. To change these values, the calling process must be either the creator of the IPC structure or the superuser. Changing these fields is similar to calling chown or chmod for a file.

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

Figure 15.24. XSI IPC permissions

❖ Configuration Limits

All three forms of XSI IPC have built-in limits that we may encounter. Most of these limits can be changed by reconfiguring the kernel. We describe the limits when we describe each of the three forms of IPC.

❖ Advantages and Disadvantages

- A fundamental problem with XSI IPC is that the IPC structures are systemwide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling msgrcv or msgctl, by someone executing the ipcrm(1) command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.
- Another problem with XSI IPC is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions. Almost a dozen new system calls (msgget, semop, shmat, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an ls command, we can't remove them with the rm command, and we can't change their permissions with the chmod command. Instead, two new commands ipcs(1) and ipcrm(1)were added.
- Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (select and poll) with them. This makes it harder to use more than one of these IPC structures at a time or to use any of these IPC structures with file or device I/O. For example, we can't have a server wait for a message to be placed on one of two message queues without some form of busywait loop.

MESSAGE QUEUES

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a queue and its identifier a queue ID.

A new queue is created or an existing queue opened by msgget. New messages are added to the end of a queue by msgsnd. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd when the message is added to a queue. Messages are fetched from a queue by msgrcv. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following msqid_ds structure associated with it:

```
struct msqid_ds
{
        struct ipc_perm  msg_perm;     /* see Section 15.6.2 */
        msgqnum_t        msg_qnum;     /* # of messages on queue */
        msglen_t         msg_qbytes;   /* max # of bytes on queue */
        pid_t            msg_lspid;    /* pid of last msgsnd() */
        pid_t            msg_lrpid;    /* pid of last msgrcv() */
```

```
          time_t          msg_stime;    /* last-msgsnd() time */
          time_t          msg_rtime;    /* last-msgrcv() time */
          time_t          msg_ctime;    /* last-change time */
          .
          .
          .
  };
```

This structure defines the current status of the queue.

The first function normally called is `msgget` to either open an existing queue or create a new queue.

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

int msgget(key_t key, int flag);
```

Returns: message queue ID if OK, 1 on error

When a new queue is created, the following members of the `msqid_ds` structure are initialized.

- ✓ The `ipc_perm` structure is initialized. The `mode` member of this structure is set to the corresponding permission bits of flag.
- ✓ `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are all set to 0.
- ✓ `msg_ctime` is set to the current time.
- ✓ `msg_qbytes` is set to the system limit.

On success, `msgget` returns the non-negative queue ID. This value is then used with the other three message queue functions.

The `msgctl` function performs various operations on a queue.

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

int msgctl(int msqid, int cmd, struct msqid_ds *buf );
```

Returns: 0 if OK, 1 on error.

The cmd argument specifies the command to be performed on the queue specified by msqid.

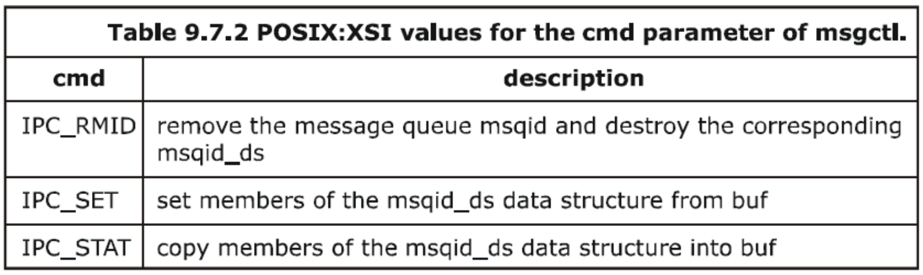


Table 9.7.2 POSIX:XSI values for the cmd parameter of msgctl.

cmd	description
IPC_RMID	remove the message queue msqid and destroy the corresponding msqid_ds
IPC_SET	set members of the msqid_ds data structure from buf
IPC_STAT	copy members of the msqid_ds data structure into buf

Data is placed onto a message queue by calling `msgsnd`.

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

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Returns: 0 if OK, 1 on error.

Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if nbytes is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
  struct mymesg
  {
    long  mtype;      /* positive message type */
    char  mtext[512]; /* message data, of length nbytes */
```

```
   };
```

The ptr argument is then a pointer to a `mymesg` structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.

Messages are retrieved from a queue by `msgrcv`.

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

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

Returns: size of data portion of message if OK, 1 on error.

The type argument lets us specify which message we want.

type == 0	The first message on the queue is returned.
type > 0	The first message on the queue whose message type equals type is returned.
type < 0	The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

SEMAPHORES

A semaphore is a counter used to provide access to a shared data object for multiple processes.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.
A common form of semaphore is called a ***binary semaphore***. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.
XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The undo feature that we describe later is supposed to handle this.

The kernel maintains a `semid_ds` structure for each semaphore set:

```
 struct semid_ds {
     struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
     unsigned short   sem_nsems; /* # of semaphores in set */
     time_t           sem_otime; /* last-semop() time */
     time_t           sem_ctime; /* last-change time */
     .
     .
     .
   };
```

Each semaphore is represented by an anonymous structure containing at least the following members:

```
 struct {
     unsigned short  semval;   /* semaphore value, always >= 0 */
     pid_t           sempid;   /* pid for last operation */
     unsigned short  semncnt;  /* # processes awaiting semval>curval */
     unsigned short  semzcnt;  /* # processes awaiting semval==0 */
     .
     .
     .
   };
```

The first function to call is semget to obtain a semaphore ID.

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

int semget(key_t key, int nsems, int flag);
```

Returns: semaphore ID if OK, 1 on error

When a new set is created, the following members of the semid_ds structure are initialized.

- The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding permission bits of flag.
- sem_otime is set to 0.
- sem_ctime is set to the current time.
- sem_nsems is set to nsems.

The number of semaphores in the set is nsems. If a new set is being created (typically in the server), we must specify nsems. If we are referencing an existing set (a client), we can specify nsems as 0.

The semctl function is the catchall for various semaphore operations.

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

int semctl(int semid, int semnum, int  cmd,... /* union semun arg */);
```

The fourth argument is optional, depending on the command requested, and if present, is of type `semun`, a union of various command-specific arguments:

```
   union semun
   {
     int   val;                /* for SETVAL */
     struct semid_ds *buf;     /* for IPC_STAT and IPC_SET */
     unsigned short  *array;   /* for GETALL and SETALL */
   };
```

Table 9.8.1 POSIX:XSI values for the cmd parameter of semctl.

cmd	description
GETALL	return values of the semaphore set in arg.array
GETVAL	return value of a specific semaphore element
GETPID	return process ID of last process to manipulate element
GETNCNT	return number of processes waiting for element to increment
GETZCNT	return number of processes waiting for element to become 0
IPC_RMID	remove semaphore set identified by semid
IPC_SET	set permissions of the semaphore set from arg.buf
IPC_STAT	copy members of semid_ds of semaphore set semid into arg.buf
SETALL	set values of semaphore set from arg.array
SETVAL	set value of a specific semaphore element to arg.val

The *cmd* argument specifies one of the above ten commands to be performed on the set specified by semid.

The function `semop` atomically performs an array of operations on a semaphore set.

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

```
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, 1 on error.

The semoparray argument is a pointer to an array of semaphore operations, represented by `sembuf` structures:

```
struct sembuf {
  unsigned short  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short           sem_op;   /* operation (negative, 0, or positive) */
  short           sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

The nops argument specifies the number of operations (elements) in the array.

The sem_op element operations are values specifying the amount by which the semaphore value is to be changed.

- If sem_op is an integer ***greater than zero***, semop adds the value to the corresponding semaphore element value and awakens all processes that are waiting for the element to increase.
- If sem_op is ***0*** and the semaphore element value is not 0, semop blocks the calling process (waiting for 0) and increments the count of processes waiting for a zero value of that element.
- If sem_op is a ***negative*** number, semop adds the sem_op value to the corresponding semaphore element value provided that the result would not be negative. If the operation would make the element value negative, semop blocks the process on the event that the semaphore element value increases. If the resulting value is 0, semop wakes the processes waiting for 0.