UNIX SYSTEM PROGRAMMING

Subject Code:10CS62 **I.A. Marks : 25**

Hours/Week : 04 **Exam Hours: 03**

Total Hours : 52 **Exam Marks: 100**

PART – A

UNIT – 1 **6 Hours**

Introduction: UNIX and ANSI Standards: The ANSI C Standard, The ANSI/ISO C++ Standards, Difference between ANSI C and C++, The POSIX Standards, The POSIX.1 FIPS Standard, The X/Open Standards.

UNIX and POSIX APIs: The POSIX APIs, The UNIX and POSIX Development Environment, API Common Characteristics.

UNIT – 2 **6 Hours**

UNIX Files: File Types, The UNIX and POSIX File System, The UNIX and POSIX File Attributes, Inodes in UNIX System V, Application Program Interface to Files, UNIX Kernel Support for Files, Relationship of C Stream Pointers and File Descriptors, Directory Files, Hard and Symbolic Links.

UNIT – 3 **7 Hours**

UNIX File APIs: General File APIs, File and Record Locking, Directory File APIs, Device File APIs, FIFO File APIs, Symbolic Link File APIs, General File Class, regfile Class for Regular Files, dirfile Class for Directory Files, FIFO File Class, Device File Class, Symbolic Link File Class, File
Listing Program.

UNIT – 4 **7 Hours**

UNIX Processes: The Environment of a UNIX Process: Introduction, main function, Process Termination, Command-Line Arguments, Environment List, Memory Layout of a C Program, Shared Libraries, Memory Allocation, Environment Variables, setjmp and longjmp Functions, getrlimit, setrlimit Functions, UNIX Kernel Support for Processes.

PART - B

UNIT – 5 **7 Hours**

Process Control : Introduction, Process Identifiers, fork, vfork, exit, wait, waitpid, wait3, wait4 Functions, Race Conditions, exec Functions, Changing User IDs and Group IDs, Interpreter Files, system Function, Process Accounting, User Identification, Process Times, I/O Redirection.

Process Relationships: Introduction, Terminal Logins, Network Logins, Process Groups, Sessions, Controlling Terminal, tcgetpgrp and tcsetpgrp Functions, Job Control, Shell Execution of Programs, Orphaned Process Groups.

UNIT – 6 **7 Hours**

Signals and Daemon Processes: Signals: The UNIX Kernel Support for Signals, signal, Signal Mask, sigaction, The SIGCHLD Signal and the waitpid Function, The sigsetjmp and siglongjmp Functions, Kill, Alarm, Interval Timers, POSIX.lb Timers.

Daemon Processes: Introduction, Daemon Characteristics, Coding Rules, Error Logging, Client-Server Model.

UNIT – 7 **6 Hours**

Interprocess Communication – 1: Overview of IPC Methods, Pipes, popen, pclose Functions, Coprocesses, FIFOs, System V IPC, Message Queues, Semaphores.

UNIT – 8 **6 Hours**

Interprocess Communication – 2: Shared Memory, Client-Server Properties, Stream Pipes, Passing File Descriptors, An Open Server-Version 1, Client-Server Connection Functions.

Text Books:

1. **Terrence Chan:** UNIX System Programming Using C++, Prentice Hall India, 1999. (Chapters 1, 5, 6, 7, 8, 9, 10)

2. **W. Richard Stevens:** Advanced Programming in the UNIX Environment, 2nd Edition, Pearson Education, 2005. (Chapters 7, 8, 9, 13, 14, 15)

Reference Books:

1. Marc J. Rochkind: Advanced UNIX Programming, 2nd Edition, Pearson Education, 2005.

2. Maurice J Bach: The Design of the UNIX Operating System, Pearson Education, 1987.

3. Uresh Vahalia: UNIX Internals: The New Frontiers, Pearson Education, 2001.

Table of contents

INTRODUCTION

1.1 UNIX AND ANSI Standards

The ISO (International Standards Organization) defines "standards are documented agreements containing technical specifications or other precise criteria to be used consistently as rules, guidelines or definitions of characteristics to ensure that materials, products, processes and services are fit for their purpose".

Most official computer standards are set by one of the following organizations:

- ANSI (American National Standards Institute)
- ITU (International Telecommunication Union)
- IEEE (Institute of Electrical and Electronic Engineers)
- ISO (International Standards Organization)
- VESA (Video Electronics Standards

1.2 The ANSI C Standard

This standard was proposed by American ANSI in the year 1989 for C programming Language standard called X3.159-1989 to standardize the C programming language constructs and libraries.

1.3 Major differences between ANSI C and K & R C

- ANSI C supports Function Prototyping
- ANSI C support of the const & volatile data type qualifier
- ANSI C support wide characters and internationalization, Defines setlocale function
- ANSI C permits function pointers to be used without dereferencing
- ANSI C defines a set of preprocessor symbols
- ANSI C defines a set of standard library functions and associated headers.

1.4 The ANSI / ISO C++ Standard

The C++ language is one of the OOP languages. It was developed by Bjarne Stroustrup at At &T Bell Laboratories. C++ is an extension of C with a major addition of the class construct features of Simula 67. The three most important facilities that C++ adds on to C are classes, function overloading, & operator overloading.

In 1989, Bjarne Stroustrup published "*The Annotated C++ Reference Manual" ,* this manual become the base for the draft ANSI C++ standard. WG21 committee of the ISO joined the ANSI X3J16 committee to develop a unify ANSI/ISO C++ standard. A draft version of ANSI/ISO standard was published in 1994.

1.5 Major Differences between ANSI and C++

- Function Declaration or Function Prototype
- Functions that take a variable number of arguments
- Type safe linkage , Linkage Directives

1.6 POSIX Standards

POSIX is acronym for Portable Operating System Interface. There are three subgroups in POSIX. They are :

POSIX.1 :

- Committee proposes a standard for base operating system APIs.
- This standard is formally known as the IEEE standard 1003.1-1990.
- This standard specifies the APIs for the file manipulation and processes (for Process Creation and Control).

POSIX.1b:

- Committee proposes a standard for real time operating system APIs
- This standard is formally known as the IEEE standard 1003.4-1993
- This standard specifies the APIs for the interprocess communication (Semaphores,Message Passing Shared Memory).

POSIX.1c:

- Committee proposes a standard for multithreaded programming interface
- This standard specifies the APIs for Thread Creation, Control, and Cleanup, Thread Scheduling,Thread Synchronization and for Signal Handling .

To ensure a user program conforms to the POSIX.1 standard, the user should define the manifested constant _POSIX_SOURCE at the beginning of each program(before the inclusion of any header files) as:

#define _ POSIX_SOURCE or

specify the –D_ POSIX_SOURCE option to a C++ compiler during compilation.

$g++ –D_ POSIX_SOURCE filename.cpp

In general a user program that must be strictly POSIX.1and POSIX.1b compliant may be written as follows:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <iostream.h>
#include <unistd.h>
int main( )
{
        ....
}
```

POSIX Feature Test Macros

Feature Test Macro	Effects if defined on a System
_POSIX_JOB_CONTROL	It allow us to start multiple jobs(groups of processes) from a single terminal and control which jobs can access the terminal and which jobs are to run in the background. Hence It supports BSD version Job Control Feature.
_POSIX_SAVED_IDS	Each process running on the system keeps the saved set-UID and set-GID, so that it can change effective user ID and group ID to those values via *setuid* and *setgid* APIs respectively.
_POSIX_CHOWN_RESTRICTED	If the defined value is -1, users may change ownership of files owned by them. Otherwise only users with special previlege may change ownership of any files on a system.
_POSIX_NO_TRUNC	If the defined value is -1, any long path name passed to an API is silently truncated to NAME_MAX bytes, otherwise error is generated.
_POSIX_VDISABLE	If the 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.

Limits Checking at Compile Time and at Run Time

- The POSIX.1 and POSIX.1b standards specify a number of parameters that describe capacity limitations of the system.
- Limits are defined in <limits.h>.
- These are prefixed with the name _POSIX _

sysconf, pathcomf and fpathconf

To find out the actual implemented configuration limits

- System wide using *sysconf* during run time
- On individual objects during run time using, *pathconf* and *fpathconf.*

```
#include <unistd.h>
long   sysconf (int  parameter);
long  fpathconf(int fildes, int flimit_name));
long  pathconf(const char *path, int  flimit_name);
```

- For pathconf(), the *path* argument points to the pathname of a file or directory.
- For fpathconf (), the *fildes* argument is an open file descriptor.

1.7 The POSIX.1 FIPS Standard

FIPS stands for Federal Information Processing Standard. This standard was developed by National Institute of Standards and Technology. The latest version of this standard, FIPS 151-1, is based on the POSIX.1- 1998 standard. The FIPS standard is a restriction of the POSIX.1-1998 standard, Thus a FIPS 151-1 conforming system is also POSIX.1-1998 conforming, but not vice versa.

FIPS 151-1 conforming system requires following features to be implemented in all FIPS conforming systems.

_POSIX_JOB_CONTROL	_POSIX_JOB_CONTROL must be defined.
_POSIX_SAVED_IDS	_POSIX_SAVED_IDS must be defined.
_POSIX_CHOWN_RESTRICTED	_POSIX_CHOWN_RESTRICTED must be defined and its value is not -1, it means users with special previlege may change ownership of any files on a system.
_POSIX_NO_TRUNC	If the defined value is -1, any long path name passed to an API is silently truncated to NAME_MAX bytes, otherwise error is generated.

_POSIX_VDISABLE	POSIX_VDISABLE must be defined and its value is not -1.
_POSIX_NO_TRUNC	Must be defined and its value is not -1, Long path name is not support.
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	
The group ID of a newly created file must inherit the group ID of its containing directory.	

Context Switching

A *user mode* is the normal execution context of any user process, and it allows the process to access its specific data only.

A *kernel mode* is the protective execution environment that allows a user process to access kernels data in a restricted manner.

When the APIs execution completes, the user process is switched back to the user mode. This context switching for each API call ensures that process access kernels data in a controlled manner and minimizes any chance of a runway user application may damage an entire system. So in general calling an APIs is more time consuming than calling a user function due to the context switching. Thus for those time critical applications, user should call their system APIs only if it is necessary.

An APIs common Characteristics

Most system calls return a special value to indicate that they have failed. The special value is typically -1, a null pointer, or a constant such as EOF that is defined for that purpose.

To find out what kind of error it was, you need to look at the error code stored in the variable errno. This variable is declared in the header file errno.h as shown below.

volatile int **errno**

- The variable errno contains the system error number.

void **perror** (*const char *message*)

- The function perror is declared in stdio.h.

Following table shows Some Error Codes and their meaning:

Errors	Meaning
EPERM	API was aborted because the calling process does not have the super user privilege.
EINTR	An APIs execution was aborted due to signal interruption.
EIO	An Input/Output error occurred in an APIs execution.
ENOEXEC	A process could not execute program via one of the Exec API.
EBADF	An API was called with an invalid file descriptor.
ECHILD	A process does not have any child process which it can wait on.
EAGAIN	An API was aborted because some system resource it is requested was temporarily unavailable. The API should call again later.
ENOMEM	An API was aborted because it could not allocate dynamic memory.
EACCESS	The process does not have enough privilege to perform the operation.
EFAULT	A pointer points to an invalid address.
EPIPE	An API attempted to write data to a pipe which has no reader.
ENOENT	An invalid file name was specified to an API.

UNIT – 2

UNIX FILES

UNIX / POSIX file Types

The different type's files available in UNIX / POSIX are:

- Regular files Example: All .exe files, C, C++, PDF Document files.
- Directory files Example: Folders in Windows.
- Device files
    - Block Device files: A physical device that transmits block of data at a time.
      For example: floppy devices CDROMs, hard disks.
    - Character Device files: A physical device that transmits data in a character based manner.
      For example: Line printers, modems etc.
- FIFO files Example: PIPEs.
- Link Files

Hard Links

It is a UNIX path or file name, by default files are having only one hard link

Symbolic Links

Symbolic links are called soft links. Soft link are created in the same manner as hard links, but it requires –s option to the *ln* command. Symbolic links are just like shortcuts in windows.

Differences between Hard links and Symbolic Links

Hard Link	Soft Links
1. Do not create new inode.	1. Create a new inode.
2. Cannot link directories unless super user privileges.	2. Can link directories.
3. Cannot link file across file systems.	3. Can link files across file systems.
4. Increase the hard link count.	4. Does not change the hard link count.
5. Always refer to the old file only,	5. Always reference to the latest

means hard links can be broken by removal of one or more links.	version of the files to which they link.

UNIX Kernel supports for file / Kernel Data structure for file manipulation

If open call succeeds, kernel establish the path between preprocess table to inode table through file table

The Steps involved in this process are:

Step 1: The kernel will search the process file descriptor table and look for first unused entry, if an entry is found, that entry will be designated to reference the file.

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

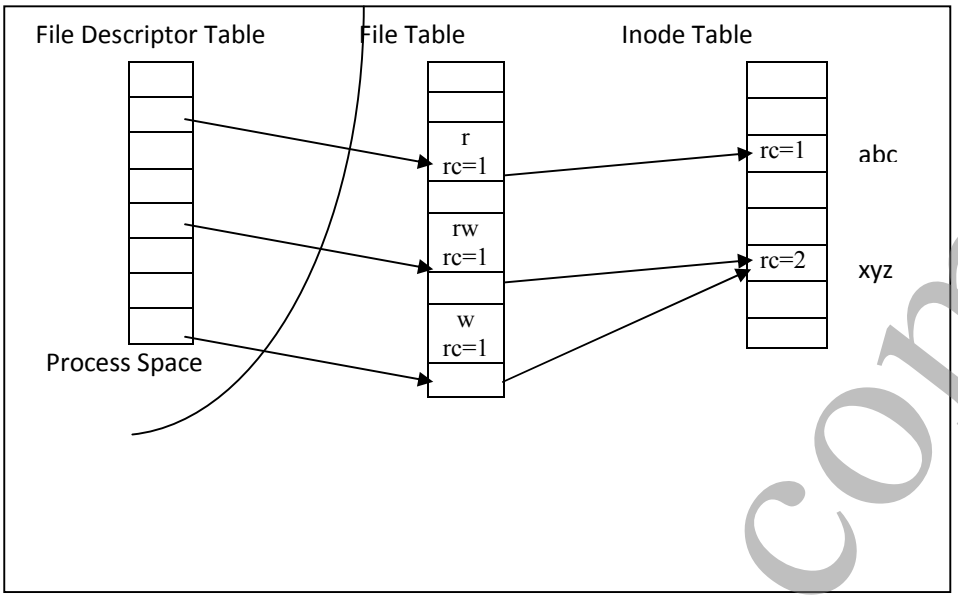
If an unused entry is found, the following events will occur.

The process's file 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.
- The file table entry will contain open mode that specifies that the file is open for read-only, write-only or read-write etc.
- The reference count in the file table entry is set to 1. The reference count keeps track of how many file descriptors from any process are referencing 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 step1 or step2 fails, the open function will return with a -1 failure status, no file descriptor table or file table entry will be allocated.

The figure shows a process's file descriptor table, the kernel file table and the inode after the process has opened three files: *abc* for read only, and *xyz* for read- write and *xyz* again for write only.



The reference count of an allocated file table entry is usually 1, but a process may

When a process calls the function *close* to close an opened file, the following sequence of events will occur.

1) The kernel sets the corresponding file descriptor table entry to be unused.
2) It decrements the reference count in the corresponding file table entry by 1. If the reference count is still non-zero, go to step 6.
3) The file table entry is marked as unused.
4) The reference count in the corresponding file inode table entry is set decremented by one. If the count is still non-zero 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 disk storage of the file.
6) It returns to the caller to the process with 0 (success) statuses.

UNIT – 3

UNIX FILE API'S

3.1 General File APIs

The file APIs that are available to perform various operations on files in a file system are:

FILE APIs	USE
open ( )	This API is used by a process to open a file for data access.
read ( )	The API is used by a process to read data from a file
write ( )	The API is used by a process to write data to a file
lseek ( )	The API is used by a process to allow random access to a file
close ( )	The API is used by a process to terminate connection to a file
stat ( ) fstat ( )	The API is used by a process to query file attributes
chmod ( )	The API is used by a process to change file access permissions.
chown ( )	The API is used by a process to change UID and/or GID of a file
utime ( )	The API is used by a process to change the last modification and access time stamps of a file
link ( )	The API is used by a process to create a hard link to a file.
unlink ( )	The API is used by a process to delete hard link of a file
umask ( )	The API is used by a process to set default file creation mask.

Open:

It is used to open or create a file by establishing a connection between the calling process and a file.

Prototype:

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

path_name : The pathname of a file to be opened or created. It can be an absolute path name or relative path name. The pathname can also be a symbolic link name.

access_mode: An integer values in the form of manifested constants which specifies how the file is to be accessed by calling process. The manifested constants can be classified as access mode flags and access modifier flags.

Access mode flags:

- **O_RDONLY:** Open the file for read only. If the file is to be opened for read only then the file should already exist in the file system and no modifier flags can be used.
- **O_WRONLY:** Open the file for write only. If the file is to be opened for write only, then any of the access modifier flags can be specified.
- **O_RDWR:** Open the file for read and write. If the file is to be opened for write only, then any of the access modifier flags can be specified.

  Access modifier flags are optional and can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file.

Access Modifier Flags:

- O_APPEND : Appends data to the end of the file. If this is not specified, data can be written anywhere in the file.
- O_CREAT : Create the file if it does not exist. If the file exists it has no effects. However if the file does not exist and O_CREATE is not specified, open will abort with a failure return status.
- O_EXCL : Used with O_CREAT, if the file exists, the call fails. The test for existence and the creation if the file does not exists.
- O_TRUNC : If the file exits, discards the file contents and sets the file size to zero.
- O_NOCTTY : Species not to use the named terminal device file as the calling process control terminal.
- O_NONBLOCK: Specifies that any subsequent read or write on the file should be non-blocking. Example, a process is normally blocked on reading an empty pipe or on writing to a pipe that is full. It may be used to specify that such read and write operations are non-blocking.

Example:

int fdesc = open("/usr/xyz/prog1", O_RDWR|O_APPEND,0);

If a file is to be opened for read-only, the file should already exist and no other modifier flags can be used.

O_APPEND, O_TRUNC, O_CREAT and O_EXCL are applicable for regular files, whereas O_NONBLOCK is for FIFO and device files only, and O_NOCTTY is for terminal device file only.

Permission:

- The permission argument is required only if the O_CREAT flag is set in the access_mode argument. It specifies the access permission of the file for its owner, group and all the other people.
- Its data type is *int* and its value is octal integer value, such as 0764. The left-most, middle and right-most bits specify the access permission for owner, group and others respectively.
- In each octal digit the left-most, middle and right-most bits specify read, write and execute permission respectively.
- For example 0764 specifies 7 is for owner, 6 is for group and 4 is for other.

  7 = 111 specifies read, write and execution permission for owner.

  6 = 110 specifies read, write permission for group.

  4 = 100 specifies read permission for others.

  Each bit is either 1, which means a right is granted or zero, for no such rights.
- POSIX.1 defines the permission data type as mode_t and its value is manifested constants which are aliases to octal integer values. For example, 0764 permission value should be specified as:

  S_IRWXU|S_IRGRP|S_IWGRP|S_IROTH
- *Permission* value is modified by its calling process *umask* value. An *umask* value specifies some access rights to be masked off (or taken away) automatically on any files created by process.
- The function prototype of the *umask* API is:

  mode_t umask (mode_t new_umask);

  It takes new mask value as argument, which is used by calling process and the function returns the old umask value. For example,

  mode_t old_mask = umask (S_IXGRP | S_IWOTH |S_IXOTH);

  The above function sets the new umask value to "no execute for group" and "no write-execute for others".

- The open function takes its permission argument value and bitwise-ANDs it with the one's complement of the calling process umask value. Thus,

  actual_permission = permission & ~umask_value

  Example: **actual_permission = 0557 & (~031) = 0546**

The return value of open function is -1 if the API fails and errno contains an error status value. If the API succeeds, the return value is file descriptor that can be used to reference the file and its value should be between 0 and OPEN_MAX-1.

Creat:

The creat system call is used to create new regular files. Its prototype is:

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

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

1. The path_name argument is the path name of a file to be created.
2. The mode argument is same as that for open API.

Since O_CREAT flag was added to open API it was used to both create and open regular files. So, the creat API has become obsolute. It is retained for backward-compatibility with early versions of UNIX.

The creat function can be implemented using the open function as:

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

read:

This function fetches a fixed size block of data from a file referenced by a given file descriptor.

Its prototype is:

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

- **fdesc:** is an integer file descriptor that refers to an opened file.
- **buf:** is the address of a buffer holding any data read.
- **size:** specifies how many bytes of data are to be read from the file.

****Note:** read function can read text or binary files. This is why the data type of buf is a universal pointer (void *). For example the following code reads, sequentially one or more record of struct sample-typed data from a file called dbase:

```
struct sample { int x; double y; char* a;} varX;
int fd = open("dbase", O_RDONLY);
while ( read(fd, &varX, sizeof(varX))>0)
```

- The return value of *read* is the number of bytes of data successfully read and stored in the *buf* argument. It should be equal to the *size* value.
- If a file contains less than *size* bytes of data remaining to be read, the return value of *read* will be less than that of *size*. If end-of-file is reached, *read* will return a zero value.
- *size_t* is defined as *int* in <sys/types.h> header, users should not set *size* to exceed INT_MAX in any *read* function call.
- If a read function call is interrupted by a caught signal and the OS does not restart the system call automatically, POSIX.1 allows two possible behaviors:
  1. The read function will return -1 value, errno will be set to EINTR, and all the data will be discarded.
  2. The read function will return the number of bytes of data read prior to the signal interruption. This allows a process to continue reading the file.
- The read function may block a calling process execution if it is reading a FIFO or device file and data is not yet available to satisfy the read request. Users may specify the O_NONBLOCK or O_NDELAY flags on a file descriptor to request nonblocking read operations on the corresponding file.

write:

The write function puts a fixed size block of data to a file referenced by a file descriptor

Its prototype is:

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

- **fdesc:** is an integer file descriptor that refers to an opened file.
- **buf:** is the address of a buffer which contains data to be written to the file.

- **size:** specifies how many bytes of data are in the buf argument.

Note: write function can read text or binary files. This is why the data type of buf is a universal pointer (void *). For example, the following code fragment writes ten records of struct sample-types data to a file called dbase2:

```
struct sample { int x; double y; char* a;} varX[10];
int fd = open("dbase2", O_WRONLY);
write(fd, (void*)varX, sizeof varX);
```

- The return value of *write* is the number of bytes of data successfully written to a file. It should be equal to the *size* value.
- If the write will cause the file size to exceed a system imposed limit or if the file system disk is full, the return value of write will be the actual number of bytes written before the function was aborted.
- If a signal arrives during a write function call and the OS does not restart the system call automatically, the write function may either return a -1 value and set errno to EINTR or return the number of bytes of data written prior to the signal interruption.
- The write function may perform nonblocking operation if the O_NONBLOCK or O_NDELAY flags are set on the fdesc argument to the function.

close:

The close function disconnects a file from a process. Its prototype is:

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

- **fdesc:** is an integer file descriptor that refers to an opened file.
- The return value of close is zero if the call succeeds or -1 if it fails.
- The close function frees unused file descriptors so that they can be reused to reference other files.
- The close function will deallocate system resources which reduces the memory requirement of a process.

- If a process terminates without closing all the files it has opened, the kernel will close files for the process.

fcntl:

The fcntl function helps to query or set access control flags and the close-on-exec flag of any file descriptor. Users can also use fcntl to assign multiple file descriptors to reference the same file. Its prototype is:

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

- **fdesc:** is an integer file descriptor that refers to an opened file.
- **cmd:** specifies which operation to perform on a file referenced by the fdesc argument.
- The third argument value, which may be specified after cmd is dependent on the actual cmd value.
- The possible cmd values are defined in the <fcntl.h> header. These values and their uses are:

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 and 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 the return value is nonzero and the flag is on. The close-on-exec flag of a newly opened file is off by default.
F_SETFD	Sets or clears the close-on-exec flag of a file descriptor fdesc. The third argument to fcntl is integer value, which is 0 to clear, or 1 to set the flag.
F_DUPFD	Duplicates the 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, in this case 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 nonblocking and in write-append mode, it can call fcntl on the file's descriptor as:

int cur_flags = fcntl(fdesc, FGETFL);

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

- The close-on-exec flag of a file descriptor specifies that if the process that owns the descriptor calls the exec API to execute different program, the fdesc should be closed by the kernel before the new program runs or not.
- The example reports the close-on-exec flag of a fdesc, sets it to on afterwards:

  cout<<fdesc<<"close-on-exec:"<<fcntl(fdesc,F_GETFD)<<endl;

  (void)fcntl(fdesc, F_SETFD, 1);
- The fcntl function can also be used to duplicate a fdesc with another fdesc. The results are two fdesc reference the same file with same access mode and share the same file pointer to read or write the file. This is useful in the redirection of standard input or output to reference a file.

  Example: Reference standard input of 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

  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_FUPFD,0)

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

  The dup function duplicates a fdesc with the lowest unused fdesc of a calling process.

  The dup2 function will duplicate a fdesc using a fd2 fdesc, regardless of whether fd2 is used to reference another file.

lseek:

The lseek system call can be used to change the file offset to a different value. It allows a process to perform random access of data on any opened file. Lseek is incompatible with FIFO files, characted device files and symbolic link files.

Its prototype is:

#include <sys/types.h>

#include <unistd.h>

off_t lseek (**int** fdesc , **off_t** pos, **int** whence);

- **fdesc:** is an integer file descriptor that refers to an opened file.
- **pos:** specifies a byte offset to be added to a reference location in deriving the new file offset value.
- **whence:** specifies 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

****NOTE:**

a. It is illegal to specify a negative pos value with the whence value set to SEEK_SET as this will set negative offset.

b. If an lseek call will result in a new file offset that is beyond end-of-file, two outcomes are possible:

1. If a file is opened for read only the lseek will fail.

2. If a file is opened for write access, lseek will succeed and it will extend the file size up to the new file offset address.

- The return value of lseek is the new file offset address where the next read of write operation will occur, or -1 if lseek call fails.

The iostream class defines *tellg* and *seekg* functions to allow users to randomly access data from any isotream class. These functions can be implemented using the *lseek* function as follows:

#include<iostream.h>

#include<sys/types.h>

#include<unistd.h>

```
streampos iostream::tellg()
{
        return (streampos)lseek(this->fileno(),(off_t)0,SEEK_CUR);
}
iostream&iostream::seekg(streampos pos,seek_dir ref_loc)
{
if(ref_loc == ios::beg)
(void)lseek(this->fileno(), (off_t)pos, SEEK_SET);
else if(ref_loc == ios::cur)
        (void)lseek(this->fileno(), (off_t)pos, SEEK_CUR);
else if(ref_loc == ios::end)
        (void)lseek(this->fileno(), (off_t)pos, SEEK_END);
return *this;
}
```

- The iostream::tellg simply calls lseek to return the current file pointer associated with an iostream object. The file descriptor of an iostream object **const char*** is obtained from the fileno member function.
- The iostream::seekg relies on lseek to alter the file pointer associated with an iostream object. The arguments are file offset and a reference location for the offset. This function also converts seek_dir value to an lseek whence value.

  There is one-to-one mapping of the seek_dir values to the whence values used by lseek:

seek_dir value	**lseek whence value**
ios::beg	SEEK_SET
ios::cur	SEEK_CUR
ios::end	SEEK_END

link:

The link function creates a new link for an existing file . This function does not create a new file. It create a new path name for an existing file. Its prototype is:

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

- **cur_link:** is a path name of an existing file.

- **new_link:** is a new path name to be assigned to the same file.
- If this call succeeds, the hard link count attribute of the file will be increased by 1.
- link cannot be used to create hard links across file systems. It cannot be used on directory files unless it is called by a process that has superuser previlege.

The *ln* command is implemented using the link API. The program is given below:

```
#include<stdio.h>
#include<unistd.h>
int main(int argc,char* argv[])
{
  if(argc!=3)
  {
                printf("usage:%s",argv[0]);
                printf("<src_file><dest_file>\n");
                return 0;
  }
  if(link(argv[1],argv[2]) == -1)
  {
                perror("link");
                return 1;
  }
  return 0;
}
```

unlink:

This function deletes a link of an existing file. It decreases the hard link count attributes of the named file, and removes the file name entry of the link from a directory file.

If this function succeeds the file can no longer be referenced by that link.

File will be removed by the file system if the hard link count of the file is zero and no process has fdesc referencing that file.

Its prototype is:

```
#include <unistd.h>

int unlink (const char* cur_link )
```

- **cur_link:** is a path name of an existing file.
- The return value is 0 if it succeeds or -1 if it fails.

- The failure can be due to invalid link name and calling process lacks access permission to remove the path name.
- It cannot be used to remove directory files unless the calling process has superuser privilege.

ANSI C defines remove function which does the similar operation of unlink. If the argument to the remove functions is empty directory it will remove the directory. The prototype of rename function is:

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

The rename will fail when the new link to be created is in a different file system than the original file.

The *mv* command can be implemented using the link and unlink APIs by the program given below:

```
#include<iostream.h>
#include<unistd.h>
#include<string.h>
int main(int argc, char* argv[])
{
        if(argc!=3 || !strcmp(argv[1],argv[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:

These functions retrieve the file attributes of a given file. The first argument of *stat* is file path name where as *fstat* is a file descriptor. The prototype is given below:

```
#include <sys/types.h>
#include <unistd.h>
int stat (const char* path_name,struct stat* statv)
int fstat (const int fdesc,struct stat* statv)
```

The second argument to stat & fstat is the address of a struct stat-typed variable. The declaration of struct stat is given below:

```
struct stat
  {
    dev_ts     t_dev; //file system ID
    ino_t      st_ino;          //File inode number
    mode_t     st_mode;         //contains file type and access flags
    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 numbers
    off_t      st_size;         //file size in number of 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 it succeeds or -1 if it fails.
- Possible failures may be that a given file path name of file descriptor is invalid, the calling process lacks permission to access the file, or the function interrupted by a signal.
  - If a path name argument specified to stat is a symbolic link file, stat will resolve the link and access the non symbolic link file. Both the functions cannot be used to obtain the attributes of symbolic link file.
  - To obtain the attributes of symbolic link file lstat function was invented. Its prototype is: int lstat (const char* path_name,struct stat* statv)

The UNIX ls command is implemented by the program given below:

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

```
static char xtbl[10] = "rwxrwxrwx";

static void display_file_type ( ostream& ofs, int st_mode )
{
    switch (st_mode &S_IFMT)
   {
         case S_IFDIR:     ofs << 'd'; return;              /* directory file */
         case S_IFCHR:     ofs << 'c'; return;              /* character device file */
         case S_IFBLK:     ofs << 'b'; return;              /* block device file */
         case S_IFREG:     ofs << ' '; return;              /* regular file */
         case S_IFLNK:     ofs << 'l'; return;              /* symbolic link file */
         case S_IFIFO:     ofs << 'p'; return;             /* FIFO file */
    }
}
/* Show access permission for owner, group, others, and any special flags */
static void display_access_perm ( ostream& ofs, int st_mode )
{
         char amode[10];
         for (int i=0, j= (1 << 8); i < 9; i++, j>>=1)
         amode[i] = (st_mode&j) ? xtbl[i] : '-';          /* set access permission */
         if (st_mode&S_ISUID) amode[2] = (amode[2]=='x') ? 'S' : 's';
         if (st_mode&S_ISGID) amode[5] = (amode[5]=='x') ? 'G' : 'g';
         if (st_mode&S_ISVTX) amode[8] = (amode[8]=='x') ? 'T' : 't';
         ofs << amode << ' ';
}
/* List attributes of one file */
static void long_list (ostream& ofs, char* path_name)
{
     struct stat            statv;
     struct group*gr_p;
     struct passwd*pw_p;
     if (lstat (path_name, &statv))
```

```
    {
        cerr<<"Invalid path name:"<< path_name<<endl;
        return;
     }
    display_file_type( ofs, statv.st_mode );
    display_access_perm( ofs, statv.st_mode );
    ofs << statv.st_nlink;                    /* display hard link count */
    gr_p = getgrgid(statv.st_gid);            /* convert GID to group name */
    pw_p = getpwuid(statv.st_uid);           /*convert UID to user name */

    ofs << ' ' <<(pw_p->pw_name ? pw_p->pw_name:statv.st_uid)
         << ' ' <<(gr_p->gr_name ? gr_p->gr_name:statv.st_gid)<< ' ';
    if ((statv.st_mode&S_IFMT) == S_IFCHR || (statv.st_mode&S_IFMT)==S_IFBLK)
        ofs << MAJOR(statv.st_rdev) << ',' <<    MINOR(statv.st_rdev);
   else ofs << statv.st_size;                          /* show file size or major/minor no. */
    ofs << ' ' << ctime (&statv.st_mtime);          /* print last modification time */
    ofs << ' ' << path_name << endl;                /* show file name */
}
/* Main loop to display file attributes one file at a time */
int main (int argc, char* argv[])
{
        if (argc==1)
            cerr << "usage: " << argv[0] << " <file path name> ...\n";
        else while (--argc >= 1) long_list( cout, *++argv);
            return 0;
}
```

access:

The access function checks the existence and/or access permission of user to a named file. The prototype is given below:

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

path_name: The pathname of a file.

flag: contains one or more of the following bit-flags.

Bit Flag	Use
F_OK	Checks whether a named file exists.
R_OK	Checks whether a calling process has read permission
W_OK	Checks whether a calling process has write permission
X_OK	Checks whether a calling process has execute permission

The flag argument value to access call is composed by bitwise-ORing one or more of the above bit-flags. The following statement checks whether a user has read and write permissions on a file /usr/sjb/file1.doc:

int rc = access("/usr/sjb/file1.doc", R_OK|W_OK);

- If a flag value is F_OK, the function returns 0 if the file exists and -1 otherwise.
  If a flag value is any combination of R_OK, W_OK and X_OK, the access function uses the calling process real user ID and real group ID to check against the file user ID and group ID. The function returns 0 if all the requested permission is permitted and -1 otherwise.

The following program uses access to determine, for each command line argument, whether a named file exists. If a named file does not exist, it will be created and initialized with a character string "Hello world".

```
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
int main(int argc, char*argv[])
{
    char buf[256];
    int fdesc,len;
    while(--argc>0)  {
        if (access(*++argv,F_OK)) {                    //a brand new file
          fdesc = open(*argv, O_WRONLY|O_CREAT, 0744);
```

```
        write(fdesc, "Hello world\n", 12);
      }
      else  {
      fdesc = open(*argv, O_RDONLY);
      while(len = read(fdesc, buf,256))
        write(1, buf, len);
            }
            close(fdesc);
  }
}
```

chmod, fchmod:

The chmod and fcmod functions change file access permissions for owner, group and others and also set-UID, set-GID and sticky flags.

A process that calls one of these functions should have the effective user ID of either the super user or the owner of the file.

The prototype of these functions is given below:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int chmod (const char* path_name, mode_t flag);
int fchmod (int fdsec, mode_t flag);
```

The chmod function uses path name of a file as a first argument whereas fchmod uses fdesc as the first argument.

The flag argument contains the new access permission and any special flags to be set on the file.

For example: The following function turns on the set-UID flag, removes group write permission and others read and execute permission on a file named /usr/sjb/prog1.c

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

```
#include <unistd.h>

void change_mode( )
{
    struct stat statv;
    int flag = (S_IWGRP|S_IROTH|S_IXOTH);
    if (stat("/usr/sjb/prog1.c", &statv))
       perror("stat");
     else {
                flag = (statv.st_mode & ~flag) | S_ISUID;
                if (chmod("usr/sjb/prog1.c", flag))
                        perror("chmod");
        }
 }
```

chown, fchown, lchown:

The chown and fchown functions change the user ID and group ID of files. They differ only in their first argument which refer to a file by either a path name or a file descriptor.

The lchown function changes the ownership of symbolic link file. The chown function changes the ownership of the file to which the symbolic link file refers.

The function prototypes of these functions are given below:

```
#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);
```

1. path_name: is the path name of a file.
2. uid: specifies the new user ID to be assigned to the file.
3. gid : specifies the new group ID to be assigned to the file.

If the actual value of uid or gid argument is -1 the ID of the file is not changed.

3.2 File and Record Locking:

UNIX systems allow multiple processes to read and write the same file concurrently which provides data sharing among processes. It also renders difficulty for any process in determining when data in a file can be overridden by another process.

In some of the applications like a database manager, where no other process can write or read a file while a process is accessing a database file. To overcome this drawback, UNIX and POSIX systems support a file locking mechanism.

File locking is applicable only for regular files. It allows a process to impose a lock on a file so that other processes cannot modify the file until it is unlocked by the process.

A process can impose a write lock or a read lock on either a portion of a file or an entire file.

The difference between write locks and read locks is that when a write lock is set, it prevents other processes from setting any overlapping read or write locks on the locked region of a file. On the other hand, when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region of a file.

The intention of a 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. A write lock is also known as an *exclusive lock*.

The use of a 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 a *shared lock.*

3.2.1 Mandatory Lock

Mandatory locks 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 data on the locked region.

If a mandatory shared lock is set on a region of a file, no process can use the *write* system call to modify the locked region.

It is used to synchronize reading and writing of shared files by multiple processes: If a process locks up a file, other processes that attempts to write to the locked regions are blocked until the former process releases its lock.

Mandatory locks may cause problems: If a runaway process sets a mandatory exclusive lock on a file and never unlocks it, no other processes can access the locked region of the file until either the runaway process is killed or the system is rebooted.

System V.3 and V.4 support mandatory locks.

3.2.2 Advisory Lock

An advisory lock is not enforced by a kernel at the system call level.

This means that even though lock (read or write) may be set on a file, other processes can still use the *read* or *write* APIs to access the file.

To make use of advisory locks, processes that manipulate the same file must cooperate such that they follow this procedure for every read or write operation to the file:

a. Try to set a lock at the region to be accessed. If this fails, a process can either wait for the lock request to become successful or go do something else and try to lock the file again later.
b. After a lock is acquired successfully, read or write the locked region release the lock
c. The drawback of advisory locks are that programs that create processes to share files must follow the above file locking procedure to be cooperative. This may be difficult to control when programs are obtained from different sources.

All UNIX and POSIX systems support advisory locks.

UNIX System V and POSIX.I use the *fcntl* API for file locking. The prototype of the *fcntl* API is:

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

The *fdesc* argument is a file descriptor for a file to be processed. The *cmd flag argu*ment defines which operation is to be performed.

cmd Flag	**Use**
F_SETLK	Sets a file lock. Do not block if this cannot succeed immediately
F_SETLKW	Sets a file lock and blocks the calling process until the lock is acquired
F_GETLK	Queries as to which process locked a specified region of a file

For file locking, the third argument to *fcntl is* an address of a *struct flock-typed* variable. This variable specifies a region of a file where the lock is to be set, unset, or queried. The *struct flock* is declared in the <fcntl.h> as:

```
struct flock
{
        short l_type;   // what lock to be set or to unlock file
        short l_whence;         // a reference address for the next field
        off_t l_start;  //offset from the l_whence reference address
        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 possible values of l_type are:

***l_ type* value**	**Use**
F_RDLCK	Sets a a read (shared) lock on a specified region
F_WRLCK	Sets a write (exclusive) lock on a specified region
F_UNLCK	Unlocks a specified region

The possible values of l_whence and their uses are:

***l_whence* value**	**Use**
SEEK_CUR	The *l_start* value is added to the current file pointer address.
SEEK_CUR	The *l_start* value is added to the current file pointer **Use** address
SEEK_SET	The *l_start* value is added to byte 0 of the file
SEEK_END	The *l_start* value ts'added to the end (current size) of the file

3.2.3 Lock Promotion and Lock splitting:

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 the process now unlocks 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.*

The procedure for setting the mandatory locks for UNIX system V3 and V4 are:

The following *file_lock.C* program illustrates a use of *fcntl* for file locking:

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

int main (int argc, cnar* argv[]) {
struct flock            fvar;
int                     fdesc;
while (--argc > 0) {                    /* do the following for each file */
 if ((fdesc=open(*++argv,O_RDWR ))==-1 ) {
perror("open"); continue;
}
fvar.l_type     = F_WRLCK;
fvar.l_whence = SEEK_SET;
fvar.l_start    = 0;
fvar.l_len      = 0;
/* Attempt to set an exclusive (write) lock on the entire file */
while (fcntl(fdesc, FSETLK,&fvar)==-1) {
/* Set lock falls, find out who has locked the file */
while (fcntl(fdesc,F_GETLK,&fvar)!=-1 && fvar.l_type != F_UNLCK){
cout<<*argv<<"locked by"<<fvar.l_pid<<"from"<<fvar.l_start<<"for"<<fvar.l_len
     <<"byte for"<<(fvar.l_type == F_WRLCK ? 'w':'r')<<endl;
if (!fvar.l_len) break;
fvar.l_start += fvar.l_len;
 fvar.l_len              = 0;
}/* while there are locks set by other processes */
} /* while set lock un-successful */
```

```
Lock the file OK. Now process data in the file */

/* Now unlock the entire file */
fvar.l_type         = F_UNLCK;
fvar.l_whence       = SEEK_SET;
fvar.l_start        = 0;
fvar.l_len          = 0;
if (fcntl(fdosc, F_SETLKW,&fear)==-1) perror("fcntl");
}
return 0;
) /* main */
```

3.3 Directory File APIs

Directory files in UNIX and POSIX systems are used to help users in organizing their files into some structure based on the specific use of file.

They are also used by the operating system to convert file path names to their inode numbers.

Directory files are created in BSD UNIX and POSIX.1 by mkdir API:

```
#include <sys/stat.h>
#include <unistd.h>
int mkdir ( const char* path_name, mode t mode );
```

1. The path_name argument is the path name of a directory to be created.
2. The mode argument specifies the access permission for the owner, group and others to be assigned to the file.
3. The return value of mkdir is 0 if it succeeds or -1 if it fails.

UNIX System V.3 uses the *mknod* API to create directory files.

UNIX System V.4 supports both the *mkdir* and *mknod APIs* for creating directory files.

The difference between the two *APIs is* that a directory created by ***mknod*** **does not contain the "."** and ".." links. On the other hand, a directory created by *mkdir* has the "." and ".." links created in one atomic operation, and it is ready to be used.

A directory file is a record-oriented file, where each record stores a file name and the mode number of a file that resides in that directory.

The following portable functions are defined for directory file browsing. These functions are defined in both the <dirent.h> and <sys/dir.h> headers.

```
#include <sys/types.h>
#if defined (BSD) && !_POSIX_SOURCE
#include <sys/dir.h>
 typedef struct direct Dirent;
#else
#include <dirent.h>
typedef struct dirent Dirent;
#endif
DIR* opendir (const char* path_name);
Dirent* readdir (DIR* dir_fdesc);
int closedir (DIR* dir_fdesc);
void rewinddir (DIR* dir_fdesc);
```

The uses of these functions are:

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.

closedir: Closes a directory file referenced by *dir_fdesc*.

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.

UNIX systems support additional functions for random access of directory file records. These functions are not supported by POSIX.1:

telldir: Returns the file pointer of a given dir_fdesc.

seekdir: Changes the file pointer of a given dir_fdesc to a specified address.

Directory files are removed by the *rmdir* API. Its prototype is given below:

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

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

```
#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 direct Dirent;
#else
#include <dirent.h>
  typedef struct dirent Dirent;
#endif

int main (int argc, char* argv[ ])
 {
Dirent* dp;
DIR*    dir_fdesc;
while (--argc > 0 ) {          /* do the following for each file */
if ( !(dir_fdesc = opendir( *++argv ) )) {
if (mkdir( *argv, S_ IRWXU|S_IRWXG|S_IRWXO) == -1 )
 perror( "opendir" );
continue;
}
/*scan each directory file twice*/
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++;                              /*count how many files in directory*/
if (!cnt) { rmdir( *argv ); break;) /* empty directory */
```

```
   rewinddir( dir fdesc );    / reset pointer for second round */
   }
   closedir( dir fdesc );
  }
}
```

3.4 Device File APIs

Device files are used to interface physical devices with application programs.

Specifically, 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 files may be character-based or block-based.

UNIX systems define the ***mknod*** API **to create device** files.

#include <sys/stat.h>

#include <unistd.h>

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

1. The path_name argument is the path name of a directory to be created.
2. The mode argument specifies the access permission for the owner, group and others to be assigned to the file.
3. The *device_id* contains the major and minor device numbers and is constructed in most UNIX systems as follows: The lowest byte of *a device_id* is set to a minor device number and the next byte is set to the major device number. For example, to create a block device file called SCSI5 with major and minor numbers of 15 and 3, respectively, and access rights of read-write-execute for everyone, the *mknod* system call is:
   mknod("SCSI5", S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO, (15<<8) 13);
4. The major and minor device numbers are extended to fourteen and eighteen bits, respectively.
5. In UNIX, if a calling process has no controlling terminal and it opens a character device file, the kernel will set this device file as the controlling terminal of the process. How-ever, if the O_NOCTTY flag is set in the *open* call, such action will be suppressed.

6. The O_NONBLOCK flag specifies that the *open* call and any subsequent *read* or *write* calls to a device file should be nonblocking to the process.

The following *test mknod.C* program illustrates 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>\n";
        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_RDWR | 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);
}
```

UNIT - 5

PROCESS CONTROL

5.1 Process identifiers

- Every process has a unique process ID, a non negative integer
- Special processes : process ID 0 scheduler process also known as swapper process ID 1 init process init process never dies ,it's a normal user process run with super user privilege process ID 2 pagedaemon

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid (void);
pid_t getppid (void);
uid_t getuid (void);
uid_t geteuid (void);
gid_t getgid (void);
gid_t getegid (void);
```

Fork function

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

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

- The new process created by fork is called child process
- The function is called once but returns twice
- The return value in the child is 0
- The return value in parent is the process ID of the new child
- The child is a copy of parent

- Child gets a copy of parents text, data , heap and stack
- Instead of completely copying we can use COW copy on write technique

```
#include<sys/types.h>
#include "ourhdr.h"
int  glob = 6;
/* external variable in initialized data */
char          buf[ ] = "a write to stdout\n";
int main(void)
{
      int       var;
      /* automatic variable on the stack */
      pid_t    pid;
      var = 88;
      if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
            err_sys("write error");
          printf("before fork\n");
      if ( (pid = fork()) < 0)
            err_sys("fork error");
      else if (pid == 0)
      {                          /* child */
            glob++;            /* modify variables */
            var++;
      }
else
            sleep(2);
      /* parent */
      printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
      exit(0);
}
```

Output

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89        child's variables were changed
pid = 429, glob = 6, var = 88        parent's copy was not changed
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

file sharing

- Fork creates a duplicate copy of the file descriptors opened by parent
- There are two ways of handling descriptors after fork

1. The parent waits for the child to complete
2. After fork the parent closes all descriptors that it doesn't need and the does the same thing

Besides open files the other properties inherited by child are

- Real user ID, group ID, effective user ID, effective group ID
- Supplementary group ID
- Process group ID
- Session ID
- Controlling terminal
- set-user-ID and set-group-ID
- 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
- Resource limits

The difference between the parent and child

- The return value of fork
- The process ID
- Parent process ID
- The values of tms_utime , tms_stime , tms_cutime , tms_ustime is 0 for child
- file locks set by parent are not inherited by child
- Pending alrams are cleared for the child
- The set of pending signals for the child is set to empty set

■ The functions of fork

1. A process can duplicate itself so that parent and child can each execute different sections of code
2. A process can execute a different program

vfork

- It is same as fork
- It is intended to create a new process when the purpose of new process is to exec a new program
- The child runs in the same address space as parent until it calls either exec or exit
- vfork guarantees that the child runs first , until the child calls exec or exit

```
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");
         if ( (pid = vfork()) < 0)
                err_sys("vfork error");
        else if (pid == 0) {                /* child */
                glob++;
        /* modify parent's variables */
                var++;
                _exit(0);                   /* child terminates */
        }
        /* parent */
        printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
        exit(0);
}
```

5.2 exit functions

- Normal termination

1. Return from main
2. Calling exit – includes calling exit handlers
3. Calling _exit – it is called by exit function

- Abnormal termination

1. Calling abort – SIGABRT
2. When process receives certain signals

- Exit status is used to notify parent how a child terminated
- When a parent terminates before the child, the child is inherited by init process
- If the child terminates before the parent then the information about the is obtained by parent when it executes wait or waitpid

- The information consists of the process ID, the termination status and amount of CPU time taken by process
- A process that has terminated , but whose parents has not yet waited for it, is called a zombie
- When a process inherited by init terminates it doesn't become a zombie
- Init executes one of the wait functions to fetch the termination status

5.3 Wait and waitpid functions

- When a child id terminated the parent is notified by the kernel by sending a SIGCHLD signal
- The termination of a child is an asynchronous event
- The parent can ignore or can provide a function that is called when the signal occurs
- The process that calls wait or waitpid can
  1. Block
  2. Return immediately with termination status of the child
  3. Return immediately with an error

  #include <sys/wait.h>

  #include <sys/types.h>

  pid_t wait (int *statloc);

  pid_t waitpid (pid_t pid,int *statloc , int options);
- Statloc is a pointer to integer
- If statloc is not a null pointer ,the termination status of the terminated process is stored in the location pointed to by the argument
- The integer status returned by the two functions give information about exit status, signal number and about generation of core file
- Macros which provide information about how a process terminated

Program to demonstrate the use 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));
}
```

WIFEXITED	TRUE – if child terminated normally WEXITSTATUS – is used to fetch the lower 8 bits of argument child passed to exit or _exit
WIFSIGNALED	TRUE – if child terminated abnormally WTERMSIG – is used to fetch the signal number that caused termination WCOREDUMP – is true is core file was generated
WIFSTOPPED	TRUE – for a child that is currently stopped WSTOPSIG -- is used to fetch the signal number that caused child to stop

5.4 Waitpid

- The interpretation of pid in waitpid depends on its value

1. Pid == -1 – waits for any child
2. Pid > 0 – waits for child whose process ID equals pid
3. Pid == 0 – waits for child whose process group ID equals that of calling process
4. Pid < -1 – waits for child whose process group ID equals to absolute value of pid

- Waitpid helps us wait for a particular process
- It is nonblocking version of wait
- It supports job control

WNOHANG	Waitpid will not block if the child specified is not available
WUNTRACED	supports job control

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.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);
}
```

5.5 Waitid

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

Returns: 0 if OK, 1 on error

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.

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.

5.6 wait3 and wait4 functions

- These functions are same as waitpid but provide additional information about the resources used by the terminated process

```
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/times.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 );
```

5.7 Race condition

- Race condition occurs when multiple process are trying to do something with shared data and final out come depends on the order in which the processes run

Program with race condition

```
#include    <sys/types.h>
#include    "ourhdr.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++; )
            putc(c, stdout);
```

```
}
/*altered program*/
#include    <sys/types.h>
#include    "ourhdr.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++; )
               putc(c, stdout);
```

```
}
```

5.8 exec functions

- Exec replaces the calling process by a new program
- The new program has same process ID as the calling process
- No new program is created , exec just replaces the current process by a new program

```
#include <unistd.h>
int exec1 ( 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[ ] );

#include        <sys/types.h>
#include        <sys/wait.h>
#include        "ourhdr.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/stevens/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);
}
```

Changing user IDs and group IDs

- Prototype

  ```
  #include <sys/types.h>
  #include <unistd.h>
  int setuid (uid_t uid);
  int setgid (gid_t gid);
  ```

- Rules

1. If the process has superuser privilege, the setuid function sets – real user ID, effective user ID , saved set-user-ID to uid
2. If the process doesnot have superuser privilege, but uid equals either real user ID or saved set-user-ID, setuid sets only effective user ID to uid
3. If neither of the two conditions is true, errno is set to EPERM and an error is returned

ID	exec	exec
	Set-user-ID bit off	Set-user-Id bit on
Real user ID	unchanged	unchanged
Effective user ID	unchanged	Set from user ID of program file
Saved set user ID	copied from effective user ID	copied from effective user ID

ID	Superuser	Unprivileged user
Real user ID	Set to uid	unchanged
Effective user ID	Set to uid	Set to uid
Saved set-user ID	Set to uid	unchanged

5.9 setreuid and setregid

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

seteuid and setegid

```
#include <sys/types.h>
#include <unistd.h>
int seteuid (uid_t euid);
int setegid (gid_t egid);
```

Interpreter files

- Files which begin with a line of the form

  #! pathname [optional argument]

  most common example : #! /bin/bash
- The actual file execed by kernel is the one specified in the pathname

```
/*example of interpreter file*/
#!/bin/awk -f
BEGIN
 {
    for (i = 0; i < ARGC; i++)
            printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

- Uses of interpreter files

1. They hide the fact that certain programs are scripts in some other language
2. They provide an efficiency gain
3. They help us write shell scripts using shells other than /bin/sh

5.10 system function

- It helps us execute a command string within a program
- System is implemented by calling fork, exec and waidpid

```
#include <stdlib.h>
 int system (const char  *cmdstring);
```

- Return values of system function
- -1 – if either fork fails or waitpid returns an error other than EINTR

- 127 -- If exec fails [as if shell has executed exit]
- termination status of shell -- if all three functions succeed

```
#include        <sys/types.h>
#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);
        }
```

/*calling system function*/

```
        #include    <sys/types.h>
        #include    <sys/wait.h>
        #include    "ourhdr.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);
        }
```

5.11 Process accounting

- Process accounting : when enabled kernel writes an accounting record each time a process terminates
- Accounting records : 32 bytes of binary data

```
Struct acct
 {
   char ac_flag;
   char ac_stat;
   uid_t ac_uid;
   gid_t ac_gid;
   dev_t ac_tty;
   time_t ac_btime;
   comp_t ac_utime;
   comp_t ac_stime;
   comp_t ac_etime;
   comp_t ac_mem;
   comp_t ac_io;
   comp_t ac_rw;
   char ac_comm;
}
```

/*prog: to generate accounting data */

```
#include<sys/types.h>
#include<sys/acct.h>
#include "ourhdr.h"
#define ACCTFILE    "/var/adm/pacct"
static unsigned long    compt2ulong(comp_t);
int main(void)
{
        struct acct             acdata;
        FILE                    *fp;
if ( (fp = fopen(ACCTFILE, "r")) == NULL)
                err_sys("can't open %s", ACCTFILE);
```

```
        while
  (fread(&acdata, sizeof(acdata), 1, fp) == 1)
 {      printf("%-*.*s  e = %6ld, chars = %7ld, "
                              "stat = %3u: %c %c %c %c\n",
                    sizeof(acdata.ac_comm),
                                  sizeof(acdata.ac_comm),
                    acdata.ac_comm,
                 compt2ulong(acdata.ac_etime),
                     compt2ulong(acdata.ac_io),
                   (unsigned char)    acdata.ac_stat,

#ifdef  ACORE
    /* SVR4 doesn't define ACORE */
         acdata.ac_flag & ACORE ? 'D' : ' ',
#else
        ' ',
#endif
#ifdef  AXSIG
             /* SVR4 doesn't define AXSIG */
                acdata.ac_flag & AXSIG ? 'X' : ' ',
#else
   ' ',
#endif
acdata.ac_flag & AFORK ? 'F' : ' ',
        acdata.ac_flag & ASU   ? 'S' : ' ');
        }
if (ferror(fp))
             err_sys("read error");
        exit(0);
}
```

```
static unsigned long
  compt2ulong(comp_t comptime)
/* convert comp_t to unsigned long */
{
        unsigned long val;
        int                             exp;
        val = comptime & 017777;
                        /* 13-bit fraction */
        exp = (comptime >> 13) & 7;
                        /* 3-bit exponent (0-7) */
        while (exp-- > 0)
                val *= 8;
        return(val);
}
```

5.12 User identification

To obtain the login name

```
#include <unistd.h>
char *getlogin (void);
```

Process times

```
#include <sys/times.h>
clock_t times (struct tms *buf);
Struct tms
 {
    clock_t tms_utime;
    clock_t tms_stime;
    clock_t tms_cutime;
    clock_t tms_cstime;
 }
```

```
#include<sys/times.h>
#include "ourhdr.h"
static void
   pr_times (clock_t, struct tms *, struct tms *);
static void       do_cmd(char *);
int main (int argc, char *argv[ ])
{           int     i;
          for (i = 1; i < argc; i++)
          do_cmd(argv[i]);
           /* once for each command-line arg */
          exit(0);
}
static void
do_cmd (char *cmd)
/* execute and time the "cmd" */
{
          struct tms        tmsstart, tmsend;
          clock_t           start, end;
          int                         status;
          fprintf(stderr, "\ncommand: %s\n", cmd);
          if ( (start = times(&tmsstart)) == -1)
          /* starting values */
                    err_sys("times error");
if ( (status = system(cmd)) < 0)
                /* execute command */
                    err_sys("system() error");
          if ( (end = times(&tmsend)) == -1)
                              /* ending values */
                    err_sys("times error");
          pr_times(end-start, &tmsstart, &tmsend);
```

```
        pr_exit(status);
}
static void
  pr_times (clock_t real, struct tms *tmsstart,
                                struct  tms *tmsend)
{  static long clktck = 0;
        if (clktck == 0)
   /* fetch clock ticks per second first time */
        if ( (clktck = sysconf(_SC_CLK_TCK)) < 0)
                err_sys("sysconf error");
        fprintf (stderr, "  real:  %7.2f\n",  real / (double) clktck);
fprintf (stderr, "   user:   %7.2f\n",(tmsend->tms_utime - tmsstart> tms_utime) /  (double)
clktck);
fprintf(stderr, "  sys:   %7.2f\n",
(tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
fprintf(stderr, "   child user:   %7.2f\n",(tmsend->tms_cutime - tmsstart-> tms_cutime) /
(double) clktck);
fprintf (stderr, "   child sys:    %7.2f\n",  (tmsend->tms_cstime - tmsstart-> tms_cstime) /
(double) clktck);
}
```

5.12 Process groups

- A process group is a collection of one or more processes.
- Each process group has a unique process group ID.
- Process group IDs are similar to process IDs---they are positive integers and they can be stored in a pid_t data type.
- The function getpgrp returns the process group ID of the calling process.
- Each process group can have a process leader. The leader is identified by having its process group ID equal its process ID.

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

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

- 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 there is at least one process in the group, regardless whether the group leader terminates or not
- process group lifetime — the period of time that begins when the group is created and ends when the last process in the group leaves the group
- A process joins an existing process group, or creates a new process group by calling setpgid.

```
#include <sys/types.h>
#include <unistd.h>
int setpgid (pid_t pid, pid_t pgid);
```

- This sets the process group ID to pgid of the process pid. If the two arguments are equal, the process specified by pid becomes a process group leader.
- A process can set the process group ID of only itself or one of its children. If *pid* is 0, the process ID of the caller is used. Also if *pgid* is 0, the process ID specified by *pid* is used as the process group ID.
- In most job-control shells this function is called after a fork to have the parent set the process group ID of the child, and to have the child set its own process group ID.

5.12 SESSIONS

- A Session is a collection of one or more groups.
- The processes in a process group are usually grouped together into a process group by a shell pipeline.
- A process establishes a new session by calling the *setsid* function.

```
#include <sys/types.h>
#include <unistd.h>
pid_t setsid (void)
```

- If the calling process is not a process group leader, this function creates a new session. Three things happen:
  1. The process becomes the session leader of this new session.
  2. 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.
  3. The process has no controlling terminal.

Controlling terminal

- characteristics of sessions and process groups
- A session can have a single controlling terminal.
- 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, then it has a single foreground process group, and all other process groups in the session are background process groups.
- Whenever we type our terminal's interrupt key or quit key this causes either the interrupt signal or the quit signal to be sent to all processes in the foreground process group.
- If a modem disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process

5.13 tcgetpgrp and tcsetpgrp Functions

- We need 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

```
#include <sys/types.h>
#include<unistd.h>
pid_t tcgetpgrp(int filedes);
int tcsetpgrp(int filedes, pid_t pgrpid);
```

- 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..

5.14 Job Control

- To allow us to start multiple jobs from a single terminal and control which jobs can access the terminal and which jobs are to be run in the background.
- It requires 3 forms of support:
  - A shell that supports job control.
  - The terminal driver in the kernel must support job control.
  - Support for certain job-control signals
  - A job is just a collection of processes, often a pipeline of processes.
  - When we start a background job, the shell assigns it a job identifier and prints one or more process IDs.
  - $ make all > Make.out &

    [1] 1475

    $ pr *.c | lpr &

    [2] 1490

    $ just press RETURN

    [2] + Done pr *.c | lpr &

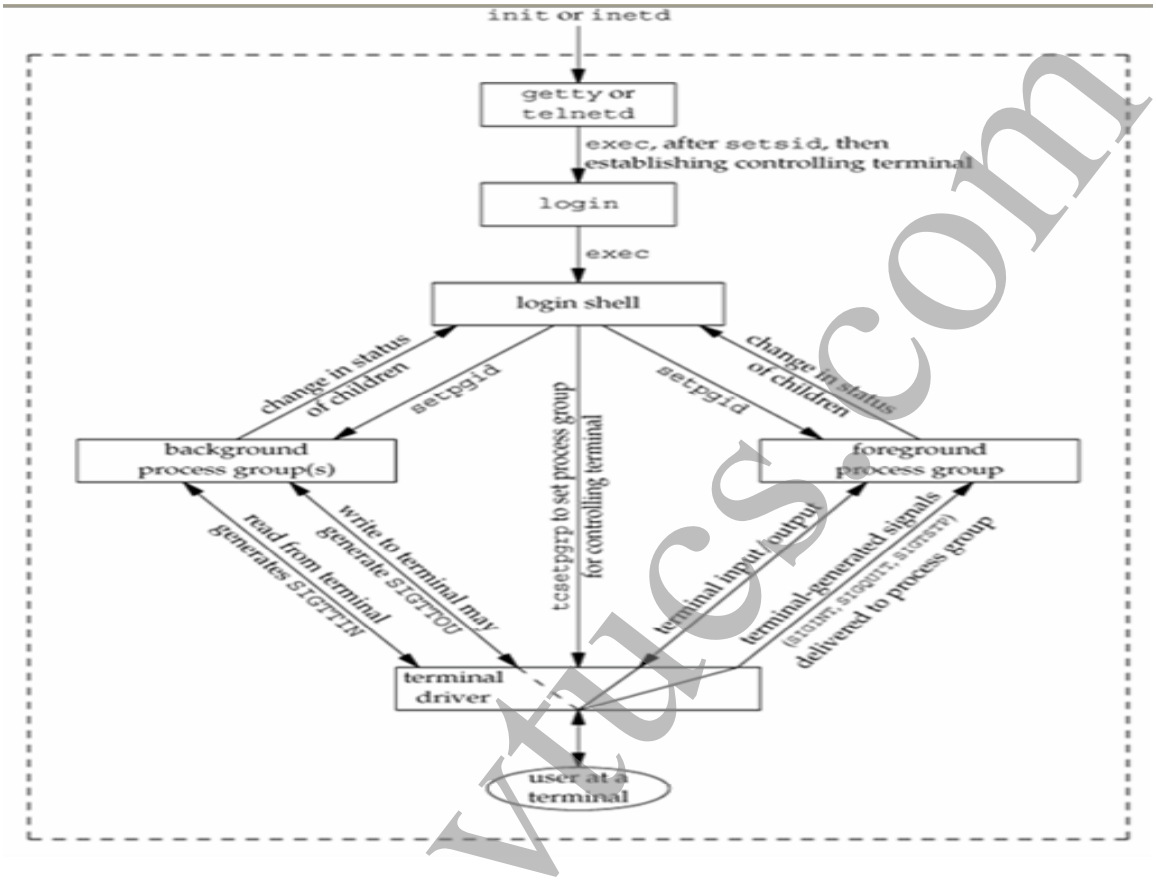
    [1] + Done make all > Make.out &

- The reason why we have to press RETURN is to have the shell print its prompt. The shell doesn't print the changed status of background jobs at any random time -- only right before it prints its prompt, to let us enter a new command line.
- Entering the suspend key (Ctrl + Z) causes the terminal driver to send the SIGTSTP signal to all processes in the foreground process group.

  The terminal driver really looks for 3 special characters, which generate signals to the foreground process group:

- The interrupt character generates SIGINT

- The quit character generates SIGQUIT
- The suspend character generates SIGTSTP



PROGRAM

$cat temp.foo & start in background, but It'll read from standard input

[1] 1681

$ we press RETURN

[1] + Stopped (tty input) cat > temp.foo &

$ fg %1 bring job number 1 to foreground

cat > temp.foo the shell tells us which job is now in the foreground

hello, world enter one line

^D type our end-of-file

$ cat temp.foo check that the one line put into the file

hello, world

- What happens if a background job outputs to the controlling terminal?
- This option we can allow or disallow. Normally we use the stty(1) command to change this option.

```
$ cat temp.foo &                    execute in background
[1]     1719
$ hello, world                      the output from the background
                                    appears after the prompt we press return
[1] + Done                          cat temp.foo &
$ stty tostop                       disable ability of background jobs to
                                    output to controlling terminal
[1]             1721
$                                   we press return and find the job is stopped
[1] + Stopped(tty output)           cat temp.foo &
```

Shell Execution Of Programs

- Bourne shell doesn't support job control
- ps –xj gives the following output

```
PPID   PID   PGID   SID   TPGID  COMMAND
   1   163   163    163   163      -sh
 163   168   163    163   163      ps
```

- Both the shell and the ps command are in the same session and foreground process group(163). The parent of the ps command is the shell.
- A process doesn't have a terminal process control group.
- A process belongs to a process group, and the process group belongs to a session. The session may or may not have a controlling terminal.
- The foreground process group ID is an attribute of the terminal, not the process.
- If ps finds that the session does not have a controlling terminal, it prints -1.

If we execute the command in the background,

```
Ps –xj &
```

The only value that changes is the process ID.

ps –xj | cat1

PPID	PID	PGID	SID	TPGID	COMMAND
1	163	163	163	163	-sh
163	200	163	163	163	cat1
200	201	163	163	163	ps

The last process in the pipeline is the child of the shell, and the first process in the pipeline is a child of the last process.

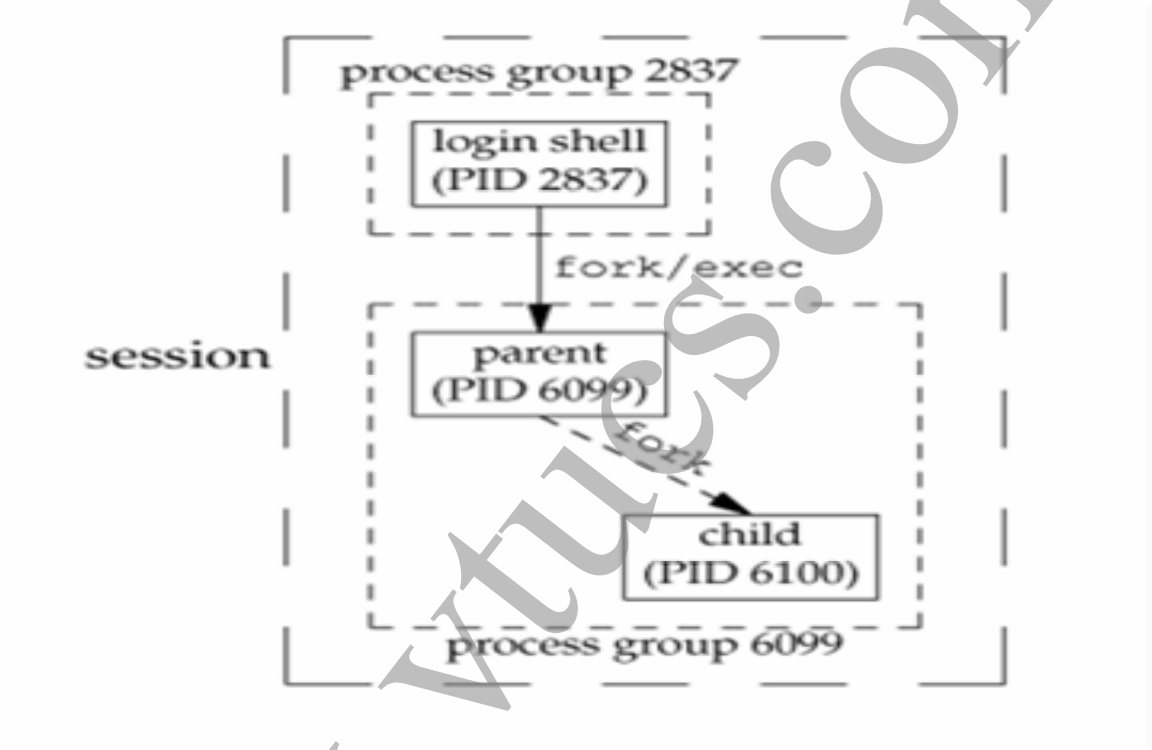
- If we execute the pipeline in the background

  ps –xj | cat1 &
- Only the process IDs change.
- Since the shell doesn't handle job control, the process group ID of the background processes remains 163, as does the terminal process group ID.

Orphaned process groups

- We know that a process whose parent terminates is called an orphan and is inherited by the init process.
- Sometimes the entire process groups can be orphaned.
- This is a job-control shell. The shell places the foreground process in its own process group(512 in the example) and the shell stays in its own process group(442). The child inherits the process group of its parent(512). After the fork,
  - The parent sleeps for 5 seconds. This is the (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 if SIGHUP is sent to the child.
  - The child itself the stop signal(SIGTSTP) with the kill function.
  - 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*.

- Since the process group is orphaned when the parent terminates, it is required that every process in the newly orphaned process group that is stopped be sent the hang-up signal (SIGHUP) followed by the continue signal.
- 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, which is why we have to provide a signal handler to catch the signal



Creating an orphaned process group

```
#include    <sys/types.h>
#include    <errno.h>
#include    <fcntl.h>
#include    <signal.h>
#include    "ourhdr.h"
static void    sig_hup(int);
static void    pr_ids(char *);
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);
     exit(0);
     }
else {                          /* child */
                pr_ids("child");
                signal(SIGHUP, sig_hup);
       /* establish signal handler */
   kill (getpid(), SIGTSTP);                    pr_ids("child");
      /* this prints only if we're continued */
                if (read(0, &c, 1) != 1)
       printf ("read error from control
            terminal,errno = %d\n", errno);             exit(0);
        }
}
static void    sig_hup (int signo)
   {
        printf("SIGHUP received, pid = %d\n",
                                        getpid());
        return;
   }
  static void  pr_ids (char *name)
   {
        printf("%s: pid = %d, ppid = %d, pgrp =
```

```
   d\n", name, getpid(), getppid(), getpgrp());
          fflush(stdout);
  }
/* OUTPUT
$ a.out
Parent: pid = 512, ppid=442, pgrp = 512
Child: parent = 513, ppid = 512, pgrp = 512
$ SIGHUP received, pid = 513
Child: pid = 513 , ppid = 1, pgrp = 512
Read error from control terminal, errno = 5
*/
```

- The parent process ID of the child has become 1.
- After calling pr_ids in the child, the program tries to read from standard input. When the background process group tries to read from its controlling terminal, SIGTTIN is generated from the background process group.
- The child becomes the background process group when the parent terminates, since the parent was executed as a foreground job by the shell

UNIT – 6

SIGNALS AND DEAMON PROCESSES

6.1. Introduction

1. Signals are triggered by events and are posted on a process to notify it that some thing has happened and requires some action.

2. Signals can be generated from a process, a user, or the UNIX kernel.

3. Example:-

    a. A process performs a divide by zero or dereferences a NULL pointer.
    b. A user hits <Delete> or <Ctrl-C> key at the keyboard.

4. A parent and child processes can send signals to each other for process synchronization.

5. Thus, signals are the software version of hardware interrupts.

6. Signals are defined as integer flags in the <signal.h> header file.

7. The following table 6.1 lists the POSIX – defined signals found in most UNIX systems.

Name	Description	Default action
SIGALRM	timer expired (alarm)	terminate
SIGABRT	abnormal termination (abort)	terminate+core
SIGFPE	arithmetic exception	terminate+core
SIGHUP	controlling terminal hangup	terminate
SIGILL	illegal machine instruction	terminate+core
SIGINT	terminal interrupt character <delete> or <ctrl-c> keys	terminate
SIGKILL	kill a process, kill -9 <pid> command.	terminate
SIGPIPE	write to pipe with no readers	terminate
SIGQUIT	terminal quit character	terminate+core
SIGSEGV	segmentation fault - invalid memory reference	terminate+core
SIGTERM	terminate process, kill <pid> command	terminate

SIGUSR1	user-defined signal	terminate
SIGUSR2	user-defined signal	terminate
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGSTOP	stop a process execution	stop process
SIGTTIN	stop a background process when it read from its control tty	stop process
SIGTSTP	stop a process execution by the ctrl-z key	stop process
SIGTTOU	stop a background process when it writes to its control tty	stop process

8. When a signal is sent to a process, it is *pending* on the process to handle it.

9. The process can react to signals in one of the three ways.

    a. Accept the default action of the signal – most signals terminate the process.
    b. Ignore the signal.
    c. Invoke a user defined function – The function is called *signal hander routine* and the signal is said to be *caught* when the function is called. If the function finishes execution without terminating the process, the process will continue execution from the point it was interrupted by the signal.

10. Most signals can be caught or ignored except the SIGKILL and SIGSTOP signals.

11. A companion signal to SIGSTOP is SIGCONT, which resumes a process execution after it has been stopped, both SIGSTOP and SIGCONT signals are used for job control in UNIX.

12. A process is allowed to ignore certain signals so that it is not interrupted while doing certain mission – critical work.

13. Example:- A DBMS process updating a database file should not be interrupted until it is finished, else database file will be corrupted, it should restore signal handling actions for signals when finished mission – critical work.

14. Because signals are generated asynchronously to a process, a process may specify a per signal handler function, these function would then be called when their corresponding signals are caught.

15. A common practice of a signal handler function is to clean up a process work environment, such as closing all input – output files, before terminating gracefully.

6.2. The UNIX Kernel Supports of Signals

1. In Unix System V.3, each entry in the kernel process table slot has an array of signal flags, one for each defined in the system.

2. 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.

3. If the recipient process is asleep (waiting a child to terminate or executing *pause* API) the kernel will awaken the process by scheduling it.

4. When the recipient process runs, the kernel will check the process U-area that contains an array of signal handling specifications, where each entry of the array corresponds to a signal defined in the system.

5. The kernel will consult the array to find out how the process will react to the pending signal.

6. If the array entry contains a zero value, the process will accept the default action of the signal, and the kernel will discard it.

7. If the array entry contains a one value, the process will ignore the signal.

8. Finally, if the array entry contains any other value, it is used as the function pointer for a used defined signal hander routine.

9. The kernel will setup the process to execute the function immediately, and the process will return to its current point of execution (or to some other place if signal hander does a long jump), if the signal hander does not terminate the process.

10. If there are different signals pending on a process, the order in which they are sent to a recipient process in undefined.

11. If multiple instances of a signal are pending on a process, it is implementation – dependent on whether a single instance or multiple instances of the signal will be delivered to the process.

12. In UNIX System V.3, each signal flag in a process table slot records only whether a signal is pending, but not how many of them are present.

6.3. signal

1. All UNIX systems and ANSI – C support the signal API, which can be used to define the per-signal handing method.

2. The function prototype of the signal is:

```
#include <signal.h>
void (*signal (int signal_num, void (*handler)(int))(int);
```

signal_num is the signal identifier like SIGINT or SIGTERM defined in the <signal.h>.

handler is the function pointer of a user defined signal handler function. This function should take an integer formal argument and does not return any value.

3. Example below attempts to catch the SIGTERM, ignores the SIGINT, and accepts the default action of the SIGSEGV signal.

4. 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>

void catch_sig(int sig_num)
{
        signal(sig_sum, catch_sig);
        cout << "catch_sig:" << sig_num << endl;
}

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

5. The SIG_IGN and SIG_DFL are manifest constants defined in <signal.h>

```
#define SIG_DFL     void (*)(int)0  // Default action
#define SIG_IGN     void (*)(int)1  // Ignore the signal
```

6. The return value of signal API is the previous signal handler for the signal.

7. UNIX system V.3 and V.4 support the sigset API, which has the same prototype and similar use a signal.

```
#include <signal.h>

void (*sigset (int signal_num, void (*handler)(int))(int);
```

the sigset arguments and return value is the same as that of signal.

8. Both the functions set signal handling methods for any named signal; but, signal API is unreliable and sigset is reliable.

9. This means that when a signal is set to be caught by a signal handler via sigset, when multiple instances of the signal arrive one of them is handled while other instances are blocked. Further, the signal handler is not reset to SIG_DFT when it is invoked.

6.4. Signal Mask

1. Each process in UNIX or POSIX.1 system has signal mask that defines which signals are blocked when generated to a process.

2. A blocked signal depends on the recipient process to unblock it and handle it accordingly.

3. If a signal is specified to be ignored and blocked, it is implementation dependent on whether the signal will be discarded or left pending when it is sent to the process.

4. 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.

5. 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);
```

new_mask defines a set of to be set or reset in a calling process signal mask.
c*md* specifies how the new_mask value is to be used by the API. The possible values cmd 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 to the calling process signal mask

6. 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.

7. The old_mask argument is the address of a *sigset_t* variable that will be assigned the calling process's original signal mask prior to a *sigprocmask* call. If the actual argument to old_mask is a NULL pointer, no previous signal mask will be returned.

8. The return value of sigprocmask call is zero if it succeeds or -1 if it fails.

9. The segset_t is a data type defined in <signal.h>. It contains a collection of bit flags, with each bit flag representing one signal defined in the system.

10. The BSD UNIX and POSIX.1 define a set of API known as *sigsetops* functions, which set, reset, and query the presence of signals in a sigset_t typed variable.

```
#include <signal.h>
int sigemptyset(sigset_t *sigmask);
int sigaddset(sigset_t *sigmask, const int signal_num);
int sigdelset(sigset_t *sigmask, const int signal_num);
int sigfillset(sigset_t sigmask);
int sigismember(const sigset_t *sigmask, const int signal_num);
```

11. The sigemptyset API clears all signal flags in the sigmask argument.

12. The sigaddset API sets the flag corresponding to the signal_num signal in sigmask.

13. The sigdelset API clears the flag corresponding to the signal_num signal in sigmask.

14. The sigfillset API sets all the flags in the sigmask.

15. The return value of the sigemptyset, sigaddset, sigdelset, and sigfillset calls is zero if the call succeed or -1 if they fail.

16. The sigismember API returns 1 if the flag corresponding to the signal_num signal in the sigmask is set, zero if not set, and -1 if the call fails.

17. 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. Then clears the SIGSEGV signal from the process signal mask.

```
#include <stdio.h>
#include <signal.h>
int main()
{
        sigset_t sigmask;
        sigemptyset(&sigmask);                  /*initialize set*/
        if (sigprocmask(0, 0, &mask) == -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");          /*set a new signal mask*/
}
```

18. If there are multiple instances of the same signal pending for the process, it is implementation dependent whether one or all of those instances will be delivered to the process.
19. A process can query which signals are pending for it via the sigpending API

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

sigmask is assigned the set of signals pending for the calling process by the API. sigpending returns a zero if it succeeds and a -1 value if it fails.

20. UNIX system V.3 and V.4 support the following APIs as simplified means 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);
```

21. The sighold API adds the named signal signal_num to the calling process signal mask.

22. The sigrelse API removes the named signal signal_num to the calling process signal mask.

23. The sigignore API sets the signal handling method for the named signal signal_num to SIG_DFT.

24. The sigpause API removes the named signal signal_num from the calling process signal mask and suspends the process until it is interrupted by a signal.

6.5 sigaction

1. The sigaction API is a replacement for the signal API in the latest UNIX and POSIX systems.

2. The sigaction API is called by a process to set up a signal handling method for each signal it wants to deal with.

3. sigaction API returns the previous signal handling method for a given signal.

4. The sigaction API prototype is:

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

5. 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;
};
```

6. The sa_handler field can be set to SIG_IGN, SIG_DFL, or a user defined signal handler function.
7. The sa_mask field specifies additional signals that process wishes to block when it is handling signal_num signal.

8. The signal_num argument designates which signal handling action is defined in the *action* argument.

9. The previous signal handling method for signal_num will be returned via the old_action argument if it is not a NULL pointer.

10. If action argument is a NULL pointer, the calling process's existing signal handling method for signal_num will be unchanged.

11. The following C program illustrates the use 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();        /* wait for signal interruption*/
        cout << argv[0] << "exits\n";
}
```

12. In the above example, the process signal mask is set with SIGTERM signal. The process then defines a signal handler for the SIGINT signal and also specifies that the SIGSEGV signal is to be blocked when the process is handling the SIGINT signal. The process then terminates its execution via the pause API.

13. The output of the above program would be as:

```
%  cc sigaction.c –o sigaction
%  ./sigaction &
[1] 495
% kill –INT 495
catch signal: 2
sigaction exits
[1]     Done            sigaction
```

14. The sa_flag field of the struct sigaction is used to specify special handling for certain signals.

15. POSIX.1 defines only two values for the sa_flag: zero or SA_NOCHLDSTOP.

16. The SA_NOCHLDSTOP flag is an integer literal defined in the <signal.h> header and can be used when signal_num is SIGCHLD.

17. The effect of the SA_NOCHLDSTOP flag is that the kernel will generate the SIGCHLD signal to a process when its child process has terminated, but not when the child process has been stopped.

18. If the sa_flag value is set to zero in sigaction call for SIGCHLD, the kernel will send the SIGCHLD signal to the calling process whenever its child process is either terminated or stopped.

19. UNIX System V.4 defines additional flags for the sa_flags field. These flags can be used to specify the UNIX System V.3 style of signal handling method:

sa_flags value	Effect on handling signal_num
SA_RESETHAND	If signal_num is caught, the sa_handler is set to SIG_DFL before the signal handler function is called, and signal_num will not be

	added to the process signal mask when the signal handler function is executed.
SA_RESTART	If a signal is caught while a process is executing a system call, the kernel will restart the system call after the signal handler returns. If this flag is not set in the sa_flags, after the signal handler returns, the system call will be aborted with a return value of -1 and will set *errno* to EINTR

6.6. The SIGCHLD Signal and the waitpid API

1. When a child process terminates or stops, the kernel will generate a SIGCHLD signal to its parent process.

2. Depending upon how the parent sets up signal handling of the SIGCHLD signal, different events may occur:

   a. Parent accepts the default action of the SIGCHLD signal: The SIGCHLD signal does not terminate the parent process. It affects only the parent process if it arrives at the same time the parent process is suspended by the waitpid system call. In this case, the parent process is awakened, the API will return child's exit status and process ID to the parent, and the kernel will clear up the process table slot allocated for the child process. Thus, with this setup, a parent process can call waitpid API repeatedly to wait for each child it created.
   b. Parent ignores the SIGCHLD signal: The SIGCHLD signal will be discarded, and the parent will not be disturbed, even if it is executing the waitpid system call. The effect of this setup is that if the parent calls waitpid API, the API will suspend the parent until all its child processes have terminated. Furthermore, the child process table slots will be cleared by the kernel, and the API will return -1 value to the parent process.

   c. Process catches the SIGCHLD signal: The signal handler function will be called in the parent whenever the child process terminates. Furthermore, if the SIGCHLD signal arrives while the parent process is executing the waitpid system call, after the signal handler returns, the waitpid API may be restarted to collect the child exit status and clear its process table slot. On the other hand, the API may be aborted and the child process table slot not freed, depending upon the parent setup of the signal action for the SIGCHLD signal.

6.7. The sigsetjmp and siglongjmp APIs

1. The sigsetjmp and siglongjmp APIs have similar functions as their corresponding setjmp and longjmp APIs.

2. The sigsetjmp and siglongjmp APIs are defined in POSIX.1 and on most UNIX systems that support signal mask.

3. The function prototypes of the APIs are:

```
#include <setjmp.h>
int sigsetjmp ( sigjmpbuf env, int save_sigmask );
int siglongjmp ( sigjmpbuf env, int ret_val );
```

4. 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.

5. The sigsetjmp API behaves similarly to the setjmp API, except that it has a second argument, *save_sigmask*, which allows a user to specify whether a calling process signal mask should be saved to the provided *env* argument.

6. If the *save_sigmask* argument is nonzero, the caller's signal mask is saved, else signal mask is not saved.

7. The siglongjmp API does all operations as the longjmp API, but it also restores a calling process signal mask if the mask was saved in its *env* argument.

8. The *ret_val* argument specifies the return value of the corresponding sigsetjmp API when called by siglongjmp API. Its value should be nonzero number, and if it is zero the siglongjmp API will reset it to 1.

9. 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 that the process signal mask is restored properly when "jumping out" from a signal handling function.

10. The following C program illustrates the use 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 ( 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 = (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\n";
                return 0;
        } else
                cerr << "Return from first time sigsetjmp is called\n";

        pause();        /* wait for signal interruption*/

}
```

11. The program sets its signal mask to contain SIGTERM, and then sets up a signal trap for the SIGINT signal.

12. The program then calls sigsetjmp to store its code location in the env global variable. **Note the sigsetjmp call returns a zero value when directly called in user program and not via siglongjmp.**

13. The program suspends its execution via the pause API.

14. When ever the user interrupts the process from the keyboard, the callme function is called.
15. The callme function calls siglongjmp API to transfer flow back to the sigsetjmp function in main, which now returns a 2 value.

16. The sample output of the above program is:

```
%  cc sigsetjmp.c
%  ./a.out &
```

```
[1] 377
Return from first time sigsetjmp is called
%  kill –INT 377
catch signal: 2
Return from signal interruption
[1]      Done            a.out
%
```

6.8. kill

1. A process can send signal to a related process via the kill API.

2. This is a simple means of IPC or control.

3. The sender and recipient processes must be related such that either sender process real or effective user ID matches that of the recipient process, or the sender has su privileges.

4. For example, a parent and child process can send signals to each other via the kill API.

5. The kill API is defined in most UNIX system and is a POSIX.1 standard.

6. The function prototype is as:

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

7. The *sig_num* argument is the integer value of a signal to be sent to one or more processes designated by *pid.*

8. The possible values of pid and its use by the kill API are:

pid value	Effects on the kill API
A positive value	*pid* is a process ID. Sends signal_num to that process.
0	Sends signal_num to all processes whose process group ID is the same as the calling process.
-1	Sends signal_num to all processes whose real user ID is the same as the effective user ID of the calling process. If the calling process effective user ID is su user ID, signal_num will be sent to all processes in the system (except processes – 0 and 1). The later case is used when the system is shutting down – kernel calls the kill API to terminate all processes except 0 and 1. **Note: POSIX.1 does not specify the behavior of the kill API when the pid value is -1. This effect is for UNIX systems only.**
A negative value	Sends signal_num to all processes whose process group ID matches the absolute value of pid.

9. The return value of kill is zero if it succeeds or -1 if it fails.

10. The following C program illustrates the implementation of the UNIX kill command.

```
#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 ) {       //get signal number
                        cerr << "Invalid number:" << argv[1] << endl;
                        return -1;
                }
                argv++; argc--;
        }
        while (--argc > 0)
                if (sscanf(*++argv, "%d", &pid) == 1) {       //get process ID
                        if ( kill ( pid, sig) == -1 )
                                perror("kill");
                } else
                        cerr << "Invalid pid:" << argv[0] << endl;
        return 0;
}
```

6.9. alarm

1. 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.

2. The alarm API is defined in most UNIX systems and is a POSIX.1 standard.

3. The function prototype of the API is as:

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

4. The *time_interval* argument is the number of CPU seconds elapse time, after which the kernel will send the SIGALRM signal to the calling process.

5. If a *time_interval* value is zero, it turns off the alarm clock.

6. The return value of the alarm API is the number of CPU seconds left in the process timer, as set by a previous alarm system call.

7. The effect of the previous alarm API call is canceled, and the process timer is reset with new alarm call.

8. A process alarm clock is not passed on to its forked child process, but an exec'ed process retains the same alarm clock value as was prior to the exec API call.

9. 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 (SIGALRM, &action, 0) == -1 ) {
                perror("sigaction");
                return -1;
        }
        (void)alarm( timer );
        (void)pause( );
}
```

10. The sleep function above sets up a signal handler for the SIGALRM, calls the alarm API to request the kernel to send the SIGALRM signal after the *timer* interval, and finally, suspends its execution via the pause system call.

11. The wakeup signal handler function is called when the SIGALRM signal is sent to the process. When it returns, the pause system call will be aborted, and the calling process will return from the sleep function.

12. BSD UNIX defines the ualarm function, which is the same as the alarm API except that the argument and return value of the ualarm function are in microsecond units.

6.10. Interval Timers

1. The use of the alarm API is to set up a interval timer in a process.

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

3. 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( SIGALRM, &action, 0) == -1) {
                perror("sigaction");
                return 1;
        }
        if ( alarm( INTERVAL ) == -1 )
                perror("alarm");
        else
                while (1) {
                        /* do normal operation */
                }
        return 0;
}
```

4. The sigaction API is called to set up callme as the signal handling function for the SIGALRM signal.

5. The program then invokes the alarm API to send itself the SIGALRM signal after 5 real clock seconds.

6. The program then goes off to perform its normal operation in an infinite loop.

7. When the timer expirers, the callme function is invoked, which restarts the alarm clock for another 5 seconds and then does the scheduled tasks.

8. When the callme function returns, the program continues its "normal" operation until another timer expiration.

9. BSD UNIX invented the setitimer API, which provides capabilities additional to those of the alarm API.

10. The *setitimer* resolution time is in microseconds, whereas the resolution time for alarm is in seconds.

11. The alarm API can be used to set up real-time clock timer per process. The setitimer API can be used to define up to three different types of timers in a process:

    a. Real time clock timer.

    b. Timer based on the user time spent by a process

    c. Timer based on the total user and system times spent by a process.

12. The getitimer API is also defined in BSD and System V UNIX for users to query the timer values that are set by the setitimer API

13. 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 );
```

14. The *which* argument specify which timer to process, the possible values are:

***which* argument value**	**Timer type**
ITIMER_REAL	Timer based on real-time clock. Generates a SIGALRM signal when expires
ITIMER_VIRTUAL	Timer based on user-time spent by a process. Generates a SIGVTALRM signal when it expires
ITIMER_PROF	Timer based on total user and system times spent by a process. Generates a SIGPROF signal when it expires

15. The *struct itimerval* data type is defined in the <sys/time.h> header as:

```
struct itimerval
{
        struct timerval it_interval;    //timer interval
        struct timerval it_value;       //current value
}
```

16. For setitimer API, the val.it_value is the time to set the named timer, and the val.it_interval is the time to reload the timer when it expires.

17. The val.it_interval may be set to zero if the timer is to run only once and if the val.it_value is set to zero, it stops the named timer if it is running.

18. For getitimer API, old.it_value and the old.it_interval return the named timer's remaining time and the reload time, respectively.

19. The old argument of the setitimer API is like the old argument of the getitimer API.

20. If this is an address of a struct itimerval typed variable, it returns the previous timer value, if set to NULL the old timer value will not be returned.

21. The ITIMER_VIRTUAL and ITIMER_PROF timers are primary useful in timing the total execution time of selected user functions, as the timer runs only while the user process is running or the kernel is executing system functions on behalf of the user process for the ITIMER_PROF timer.

22. Both the APIs return zero on success or -1 value if they fail.

23. Timers set by the setitimer API in a parent process are not inherited by its child processes, but these timers are retained when a process exec's a new program.

24. The following program illustrates the use of setitimer API.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/time.h>
#define INTERVAL   2

void callme ( int sig_no )
{
        /* do some 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( SIGALRM, &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("setitimer");
        else
                while (1) {
                        /* do normal operation */
                }
        return 0;
}
```

25. Since the setitimer and alarm APIs require that users set up signal handling to catch timer expiration, they should not be used in conjunction with the sleep API, because sleep API may modify the signal handling function for the SIGALRM signal.

6.11. POSIX.1b Timers

1. POSIX.1b defines a set of APIs for interval timer manipulation.

2. The POSIX.1b timers are more flexible and powerful than UNIX timers in following ways:

    a. Users may define multiple independent timers per system clock.

    b. The timer resolution is in nanoseconds.

    c. Users may specify, on a timer basis, the signal to be raised when a timer expires.

    d. The timer interval may be specified as either an absolute or a relative time.

3. There is a limit on how many POSIX timers can be created per process, this is TIMER_MAX constant defined in <limits.h> header.

4. POSIX timers created by a process are not inherited by its child process, but are retained across the exec system call.

5. A POSIX.1 timer does not use the SIGALRM signal when it expires, it can be used safely with the sleep API in the same program.

6. The POSIX.1b APIs for timer manipulation 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 itimrspec *val, struct itimrspec
*old);
int timer_gettime(timer_t timer_hdr, struct itimrspec *old);
int timer_getoverrun(timer_t timer_hdr);
int timer_delete(timer_t timer_hdr);
```

7. The timer_create API is used to dynamically create a timer and returns its handler.

8. The clock argument specifies which system clock would be the new timer based on, its value may be CLOCK_REALTIME for creating a real time clock timer – this defined by POSIX.1b – other values are system dependent.

9. The spec argument defines what action to take when the timer expires. The struct sigevent data type is defined as:

```
struct sigevent
{
        int             sigev_notify;
        int             sigev_signo;
        union sigval    sigev_value;
};
```

10. The sigev_signo field specifies a signal number to be raised at the timer expiration. Its valid only when the sigev_notify field is set to SIGEV_SIGNAL.

11. If sigev_notify field is set to SIGEV_NONE, no signal is raised by the timer when it expires.

12. Because multiple timers may generate the same signal, the sigev_value field is used to contain any user defined data to identify that a signal is raised by a specific timer. The data structure of the sigev_field is:

```
union sigval
{
```

```
        int         sival_int;
        void        *sival_ptr;
}
```

13. For example, a process may assign each timer a unique integer ID assigned to the *spec→sigev_value.sival_int* field
14. To pass this data along with the signal (*sigev_signo*) when it is raised, the SA_SIGINFO flag should be set in an *sigaction* call, which sets up the handling for the signal and the handling function prototype should be :

    void <sihnal handler> (int signo, siginfo_t* evp, void *ucontext);

    when the signal handler is called, the evp→si_value contains the data of the *spec→sigev_value.* The siginfo_t data type is defined in <siginfo.h>

15. If spec is set to NULL and the timer is based on CLOCK_REALTIME, then SIGALRM signal is raised when the timer expires.

16. Finally the *timer_hdrp* argument is an address of a timer_t typed variable to hold the handler of the newly generated timer. This should not be NULL as it is used to call other POIXS.1b timer APIs.

17. All POSIX.1b timer APIs return zero on success and -1 if they fail.

18. The timer_settime starts and stops a timer running. The timer_gettime is used to query the current values of the timer.

19. The struct itimerspec data type is defined as:

```
struct itimerspec
{
        struct timerspec        it_interval;
        struct timerspec        it_value;
};
```

    and the struct timerspec is defined as:

```
struct timerspec
{
        time_t      tv_sec;
        long        tv_nsec;
};
```

20. The itimerspec.it_value specifies the time remaining in the timer, and itimerspec.it_interval specifies the new time to reload the timer when it expires. All

times are specified in seconds via timerspec.tv_sec and in nanoseconds via timerspec.tv_nsec.

21. The flag parameter in timer_settime may be 0 or TIMER_REALTIME if the timer start time (val argument) is relative to the current time.

22. If the flag value is TIMER_ABSTIME, the timer start time is an absolute time.

23. The old argument is used to obtain previous timer values, if NULL no timer values are returned.

24. The old argument of timer_gettime returns the current values of named timer.

25. The timer_getoverrun API returns the number of signals generated by a timer but was lost (overrun). Timer signals are not queued by the kernel if they are raised but not processed by target process, instead the kernel records the number of overrun signals per timer.

26. The timer_destroy is used to destroy a timer created by timer_create API.

6.12. Daemon Processes: Introduction

1. 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.

2. They do not have a controlling terminal; so we say that they run in the background. UNIX systems have numerous daemons that perform day-to-day activities.

3. Here we look at the process structure of daemons and how to write a daemon.

4. Since a daemon does not have a controlling terminal, we need to see how a daemon can report error conditions when something goes wrong.

6.13. Daemon Characteristics

1. We look at some common system daemons and how they relate to the concepts of process groups, controlling terminals, and sessions.

2. The ps command prints the status of various processes in the system. We will execute: ***ps -axj*** **under BSD UNIX**

3. The -a option shows the status of processes owned by others, and -x shows processes that do not have a controlling terminal. The -j option displays the job-related information: the session ID, process group ID, controlling terminal, and terminal process group ID.

4. Under System V based systems, a similar command is *ps -efjc*.

5. The output from ps looks like

PPID	PID	PGID	SID	TTY	TPGID	UID	COMMAND
0	1	0	0	?	-1	0	init
1	2	1	1	?	-1	0	[keventd]
1	3	1	1	?	-1	0	[kapmd]
0	5	1	1	?	-1	0	[kswapd]
0	6	1	1	?	-1	0	[bdflush]
0	7	1	1	?	-1	0	[kupdated]
1	1009	1009	1009	?	-1	32	portmap
1	1048	1048	1048	?	-1	0	syslogd -m 0
1	1335	1335	1335	?	-1	0	xinetd -pidfile /var/run/xinetd.pid
1	1403	1	1	?	-1	0	[nfsd]
1	1405	1	1	?	-1	0	[lockd]
1405	1406	1	1	?	-1	0	[rpciod]
1	1853	1853	1853	?	-1	0	crond
1	2182	2182	2182	?	-1	0	/usr/sbin/cupsd

6. The system processes depend on the operating system implementation. Anything with a parent process ID of 0 is usually a kernel process started as part of the system bootstrap procedure. (An exception to this is init, since it is a user-level command started by the kernel at boot time.)

7. Kernel processes are special and generally exist for the entire lifetime of the system. They run with superuser privileges and have no controlling terminal and no command line.

8. Process 1 is usually init, is a system daemon responsible for, among other things, starting system services specific to various run levels. These services are usually implemented with the help of their own daemons.

9. On Linux, the kevenTD daemon provides process context for running scheduled functions in the kernel.

10. The kapmd daemon provides support for the advanced power management features available with various computer systems.

11. The kswapd daemon is also known as the pageout daemon. It supports the virtual memory subsystem by writing dirty pages to disk slowly over time.

12. The Linux kernel flushes cached data to disk using two additional daemons: bdflush and kupdated.

13. The portmapper daemon, portmap, provides the service of mapping RPC (Remote Procedure Call) program numbers to network port numbers.

14. The syslogd daemon is available to any program to log system messages for an operator. The messages may be printed on a console device and also written to a file.

15. The inetd daemon (xinetd) listens on the system's network interfaces for incoming requests for various network servers.

16. The nfsd, lockd, and rpciod daemons provide support for the Network File System (NFS).

17. The cron daemon (crond) executes commands at specified dates and times. Numerous system administration tasks are handled by having programs executed regularly by cron.

18. The cupsd daemon is a print spooler; it handles print requests on the system.

19. The kernel daemons are started without a controlling terminal. The lack of a controlling terminal in the user-level daemons is probably the result of the daemons having called setsid.

20. All the user-level daemons are process group leaders and session leaders and are the only processes in their process group and session. Finally, note that the parent of most of these daemons is the init process.

6.14. Coding Rules

1. Some basic rules to coding a daemon prevent unwanted interactions from happening.

2. We state these rules and then show a function, *daemonize,* that implements them.

   a. The first thing to do is call umask to set the file mode creation mask to 0. The file mode creation mask that is inherited could be set to deny certain permissions.

   b. 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 are guaranteed that the child is not a process group leader. This is a prerequisite for the call to setsid that is done next.

c. Call setsid to create a new session. Three steps occur. The process
- becomes a session leader of a new session,
- becomes the process group leader of a new process group
- has no controlling terminal.

d. 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.

e. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent.

f. 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

3. The function below can be called from a program that wants to initialize itself as a daemon.

```
#include <stdio.h>
#include <unistd.h>
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>

void daemonize(const char *cmd)
{
   int              i, fd0, fd1, fd2;
   pid_t            pid;
   struct rlimit    rl;
   struct sigaction sa;
```

```
/* Clear file creation mask. */
umask(0);

/* Get maximum number of file descriptors. */
if (getrlimit(RLIMIT_NOFILE, &rl) < 0) {
    printf("%s: can't get file limit", cmd);
    exit(1);
}


/* Become a session leader to lose controlling TTY. */
if ((pid = fork()) < 0) {
    printf("%s: can't fork", cmd);
    exit(1);
}
else if (pid != 0)          /* parent */
    exit(0);
setsid();

/* Ensure future opens won't allocate controlling TTYs.  */
sa.sa_handler = SIG_IGN;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, NULL) < 0) {
    perror("sigaction: can't ignore SIGHUP");
    exit(1);
}
if ((pid = fork()) < 0) {
    printf("%s: can't fork", cmd);
    exit(1);
}
else if (pid != 0) /* parent */
    exit(0);
/* Change the current working directory to the root so
 * we won't prevent file systems from being unmounted.  */
if (chdir("/") < 0) {
    perror("chdir: can't change directory to /");
    exit(1);
}
/* Close all open file descriptors. */
if (rl.rlim_max == RLIM_INFINITY)
    rl.rlim_max = 1024;
for (i = 0; i < rl.rlim_max; i++)
    close(i);
/* Attach file descriptors 0, 1, and 2 to /dev/null. */
fd0 = open("/dev/null", O_RDWR);
```

```
    fd1 = dup(0);
    fd2 = dup(0);
    /* Initialize the log file. */
    openlog(cmd, LOG_CONS, LOG_DAEMON);
    if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
        syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
          fd0, fd1, fd2);
        exit(1);
    }
}
```

4. If the daemonize function is called from a main program that then goes to sleep, we can check the status of the daemon with the ps command:

```
$ ./a.out
$ ps -axj
 PPID    PID     PGID    SID    TTY    TPGID  UID    COMMAND
   1     3346    3345    3345    ?       -1     501    ./a.out
```

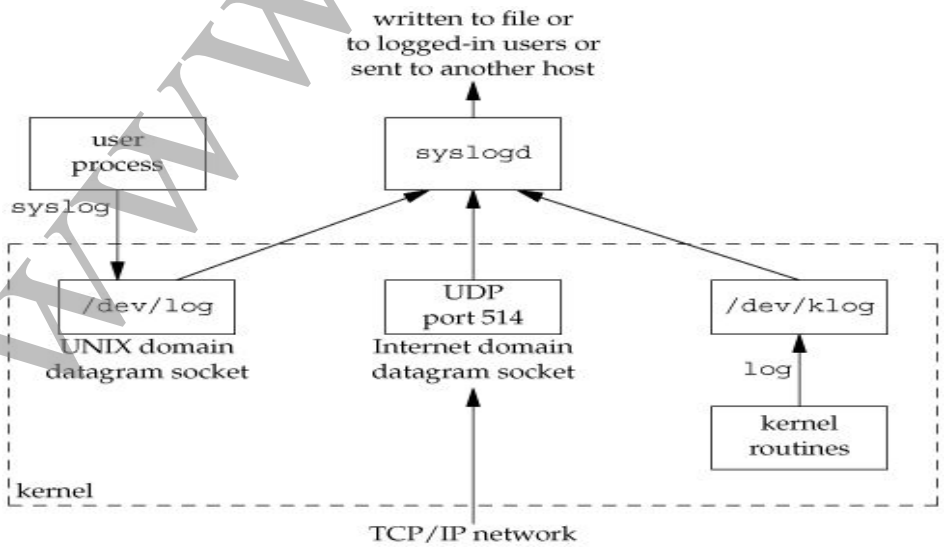
6.15. Error Logging

1. One problem a daemon has is how to handle error messages. It can not simply write to standard error, since it should not have a controlling terminal.

2. The BSD syslog facility is in 4.2BSD and most systems derived from BSD support syslog.

3. The syslog function is included as an XSI extension in the Single UNIX Specification.

4. The BSD syslog facility is used by most daemons. Figure below illustrates its structure.



5. There are three ways to generate log messages:

   a. Kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device.
   b. Most user processes (daemons) call the syslog function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
   c. 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.

6. Normally, the syslogd daemon reads all three forms of log messages.

7. On start-up, this daemon reads a configuration file, usually /etc/syslog.conf, which determines where different classes of messages are to be sent.

8. 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.

9. 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);
```

10. Calling openlog is optional. If it's not called, the first time syslog is called, openlog is called automatically.

11. Calling closelog is also optional—it just closes the descriptor that was being used to communicate with the syslogd daemon.

12. Calling openlog lets us specify an ident that is added to each log message. This is normally the name of the program (cron, inetd, etc.).

13. The option argument is a bitmask specifying various options.

14. The available options are as follows:

option	Description
LOG_CONS	If the log message can't be sent to syslogd via the UNIX domain datagram, the message is written to the console instead.
LOG_NDELAY	Open the UNIX domain datagram socket to the syslogd daemon immediately; don't wait until the first message is logged. Normally, the

option	Description
	socket is not opened until the first message is logged.
LOG_NOWAIT	Do not wait for child processes that might have been created in the process of logging the message. This prevents conflicts with applications that catch SIGCHLD, since the application might have retrieved the child's status by the time that syslog calls wait.
LOG_ODELAY	Delay the open of the connection to the syslogd daemon until the first message is logged.
LOG_PERROR	Write the log message to standard error in addition to sending it to syslogd. (Unavailable on Solaris.)
LOG_PID	Log the process ID with each message. This is intended for daemons that fork a child process to handle different requests (as compared to daemons, such as syslogd, that never call fork).

15. The facility argument for openlog is takes on the following values.

facility	Description
LOG_AUTH	authorization programs: login, su, getty, ...
LOG_AUTHPRIV	same as LOG_AUTH, but logged to file with restricted permissions
LOG_CRON	Cron and at
LOG_DAEMON	system daemons: inetd, routed, ...
LOG_FTP	the FTP daemon (ftpd)
LOG_KERN	messages generated by the kernel
LOG_LOCAL0	Reserved for local use
LOG_LOCAL1	Reserved for local use
LOG_LOCAL2	Reserved for local use
LOG_LOCAL3	Reserved for local use
LOG_LOCAL4	Reserved for local use
LOG_LOCAL5	Reserved for local use
LOG_LOCAL6	Reserved for local use

facility	Description
LOG_LOCAL7	Reserved for local use
LOG_LPR	line printer system: lpd, lpc, ...
LOG_MAIL	the mail system
LOG_NEWS	the Usenet network news system
LOG_SYSLOG	the syslogd daemon itself
LOG_USER	messages from other user processes (default)
LOG_UUCP	the UUCP system

16. The reason for the facility argument is to let the configuration file specify that messages from different facilities are to be handled differently. If we don't call openlog, or if we call it with a facility of 0, we can still specify the facility as part of the priority argument to syslog.

17. The syslog is called to generate a log message.

18. The priority argument is a combination of the facility listed above and and a level, listed below. These levels are ordered by priority, from highest to lowest.

level	Description
LOG_EMERG	emergency (system is unusable) (highest priority)
LOG_ALERT	condition that must be fixed immediately
LOG_CRIT	critical condition (e.g., hard device error)
LOG_ERR	error condition
LOG_WARNING	warning condition
LOG_NOTICE	normal, but significant condition
LOG_INFO	informational message
LOG_DEBUG	debug message (lowest priority)

19. The format argument and any remaining arguments are passed to the vsprintf function for formatting.

20. Any occurrence of the two characters %m in the format are first replaced with the error message string (strerror) corresponding to the value of errno.

21. The setlogmask function can be used to set the log priority mask for the process. This function returns the previous mask.

22. When the log priority mask is set, messages are not logged unless their priority is set in the log priority mask. Note that attempts to set the log priority mask to 0 will have no effect.

23. The logger program is also provided by many systems as a way to send log messages to the syslog facility. This logger command is intended for a shell script running none interactively that needs to generate log messages.

Example

24. In a (hypothetical) line printer spooler daemon, you might encounter the sequence

```
openlog("lpd", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

25. The first call sets the ident string to the program name, specifies that the process ID should always be printed, and sets the default facility to the line printer system.

26. The call to syslog specifies an error condition and a message string.

27. If we had not called openlog, the second call could have been

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

Here, we specify the priority argument as a combination of a level and a facility.

28. In addition to syslog, many platforms provide a variant that handles variable argument lists.

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog(int priority, const char *format, va_list arg);
```

29. Most syslogd implementations will queue messages for a short time. If a duplicate message arrives during this time, the syslog daemon will not write it to the log.

30. Instead, the daemon will print out a message similar to "last message repeated N times."

6.16. Single-Instance Daemons

1. Some daemons are implemented so that only a single copy of the daemon should be running at a time for proper operation.

2. The daemon might need exclusive access to a device, for example. In the case of the cron daemon, if multiple instances were running, each copy might try to start a single scheduled operation, resulting in duplicate operations and probably an error.

3. If the daemon needs to access a device, the device driver will sometimes prevent multiple opens of the corresponding device node in /dev.

4. This restricts us to one copy of the daemon running at a time. If no such device is available, however, we need to do the work ourselves.

5. The file- and record-locking mechanism provides the basis for one way to ensure that only one copy of a daemon is running.

6. 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.

7. Successive attempts to create write locks will fail, serving as an indication to successive copies of the daemon that another instance is already running.

8. File and record locking provides a convenient mutual-exclusion mechanism.

9. 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.

Example

10. The function shown below illustrates the use of file and record locking to ensure that only one copy of a daemon is running.

11. Each copy of the daemon will try to create a file and write its process ID in it. This will allow administrators to identify the process easily.

12. If the file is already locked, the lockfile function will fail with errno set to EACCES or EAGAIN, so we return 1, indicating that the daemon is already running. Otherwise, we truncate the file, write our process ID to it, and return 0.

13. Truncating the file prevents data from the previous daemon appearing as if it applies to the current daemon.

```
#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);
}
```

6.17. Daemon Conventions

1. Several common conventions are followed by daemons in the UNIX System.

   - 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.

6.18. ClientServer Model

1. A common use for a daemon process is as a server process.

2. We can call the syslogd process a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.

3. In general, a server is a process that waits for a client to contact it, requesting some type of service.

4. The service being provided by the syslogd server is the logging of an error message.

UNIT – 7

INTERPROCESS COMMUNICATION

7.1. Introduction

1. We have seen the process control primitives and saw how to invoke multiple processes.

2. The only way for these processes to exchange information is by passing open files across a fork or an exec or through the file system.

3. We will now describe other techniques for processes to communicate with each other: IPC, or inter-process communication.

4. In the past, UNIX System IPC was a hodgepodge of various approaches, few of which were portable across all UNIX system implementations.

5. Through the POSIX and The Open Group (formerly X/Open) standardization efforts, the situation has improved, but differences still exist.

6. Table 7.1 summarizes the various forms of IPC.

Sno.	IPC type
1.	half-duplex pipes
2.	FIFOs
3.	full-duplex pipes
4.	named full-duplex pipes
5.	message queues
6.	Semaphores
7.	shared memory
8.	Sockets
9	STREAMS

7. The first seven forms of IPC in Table 7.1 are usually restricted to IPC between processes on the same host.

8. The final two rows—sockets and STREAMS—are the only two that are generally supported for IPC between processes on different hosts.

7.2. Pipes

1. Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

   a.) Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.

   b.) 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.

2. Half-duplex pipes are the most commonly used form of IPC.
3. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one to the standard input of the next using a pipe.

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

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

Returns: 0 if OK, -1 on error

5. Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing.

6. The output of filedes[1] is the input for filedes[0].

7. Two ways to picture a half-duplex pipe are shown in figure 7.1.

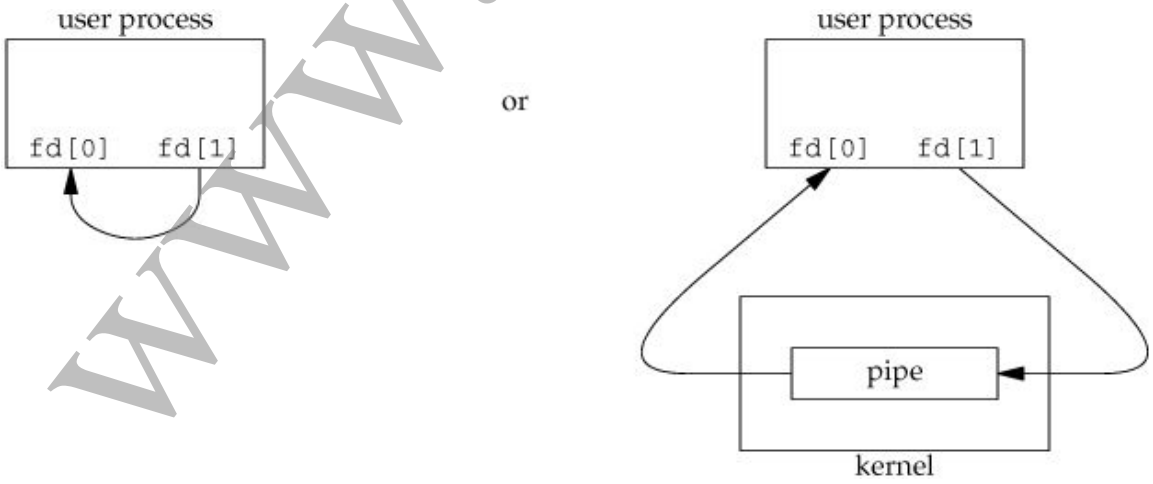


Figure 7.1: Two ways to view a half-duplex pipe

8. The left half of the figure 7.1 shows the two ends of the pipe connected in a single process.

9. The right half of the figure 7.1 emphasizes that the data in the pipe flows through the kernel.

10. The fstat function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the S_ISFIFO macro.

11. A pipe in a single process is next to useless.

12. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa.

13. What happens after the fork depends on which direction of data flow we want.

14. 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]).

15. Figure 7.2 shows the resulting arrangement of descriptors.

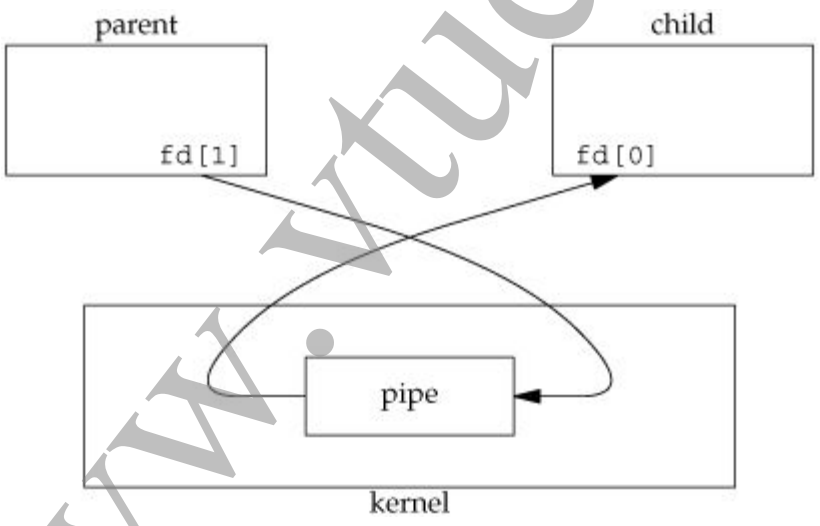


Figure 7.2: Pipe from parent to child

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

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

    1. 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.

2. 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.

18. When we are writing to a pipe (or FIFO), the constant PIPE_BUF specifies the kernel's pipe buffer size.

Example

19. Program 7.1 shows the code to create a pipe between a parent and its child and to send data down the pipe.

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

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

   if (pipe(fd) < 0)
      perror("pipe error");
   if ((pid = fork()) < 0) {
      perror("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);
}
```

Program 7.1: Send data from parent to child over a pipe

20. In Program 7.1, we called read and write directly on the pipe descriptors.

21. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output.

22. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

Example

23. Recall the five functions TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT, and WAIT_CHILD.

24. Program code 7.2 shows an implementation of the functions using pipes.

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

static int  pfd1[2], pfd2[2];
void TELL_WAIT(void)
{
   if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
      perror("pipe error");
}
void TELL_PARENT(pid_t pid)
{
   if (write(pfd2[1], "c", 1) != 1)
      perror("write error");
}
void WAIT_PARENT(void)
{
   char    c;
   if (read(pfd1[0], &c, 1) != 1)
      perror("read error");
   if (c != 'p') {
      printf("WAIT_PARENT: incorrect data");
      exit(1);
   }
}
void TELL_CHILD(pid_t pid)
{
   if (write(pfd1[1], "p", 1) != 1)
      perror("write error");
}
void WAIT_CHILD(void)
{
   char    c;
   if (read(pfd2[0], &c, 1) != 1)
      perror("read error");
```

```
    if (c != 'c')
        printf("WAIT_CHILD: incorrect data");
        exit(1);
    }
}
```

Program 7.2: Routines to let a parent and child synchronize

25. We create two pipes before the fork, as shown in program 7.2.

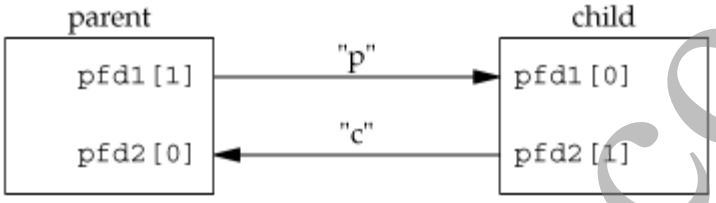


Figure 7.3: Using two pipes for parent—child synchronization

26. The parent writes the character "p" across the top pipe when TELL_CHILD is called, and the child writes the character "c" across the bottom pipe when TELL_PARENT is called.

27. The corresponding WAIT_xxx functions do a blocking read for the single character.

28. Note that each pipe has an extra reader, which doesn't matter. That is, in addition to the child reading from pfd1[0], the parent also has this end of the top pipe open for reading. This does not affect us, since the parent doesn't try to read from this pipe.

7.3. popen and pclose Functions

1. 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 provides the popen and pclose functions.

2. These two functions handle all the dirty work that we have been doing ourselves:

   - Creating a pipe
   - Forking a child
   - Closing the unused ends of the pipe
   - Executing a shell to run the command
   - And waiting for the command to terminate.

3. The prototypes of these functions are:

```
#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

4. The function popen does a fork and exec to execute the cmdstring, and returns a standard I/O file pointer.
5. If argument type is "r", the file pointer is connected to the standard output of cmdstring (figure 7.4)

Figure 7.4: Result of fp = popen(cmdstring, "r")

6. If type is "w", the file pointer is connected to the standard input of cmdstring, as shown in figure 7.5.

Figure 7.5. Result of fp = popen(cmdstring, "w")

7. The pclose function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell.
8. If the shell cannot be executed, the termination status returned by pclose is as if the shell had executed exit(127).
9. The cmdstring is executed by the Bourne shell, as in

```
sh -c cmdstring
```

10. This means that the shell expands any of its special characters in cmdstring. This allows us to say, for example,

```
fp = popen("ls *.c", "r");
```

or

```
fp = popen("cmd 2>&1", "r");
```

Example: Implementation of popen and pclose Functions

11. Program code 7.3 shows the implementation of popen and pclose.

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

/* Pointer to array allocated at run-time. */
static pid_t    *childpid = NULL;
static int      maxfd;

FILE * popen(const char *cmdstring, const char *type)
{
   int     i;
   int     pfd[2];
   pid_t   pid;
   FILE    *fp;
   /* only allow "r" or "w" */
   if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
      errno = EINVAL;     /* required by POSIX */
      return(NULL);
   }

   if (childpid == NULL) {     /* first time through */
      /* allocate zeroed out array for child pids */
      maxfd = sysconf(_SC_OPEN_MAX);
      if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
         return(NULL);
   }
   if (pipe(pfd) < 0)
      return(NULL);   /* errno set by pipe() */

   if ((pid = fork()) < 0) {
      return(NULL);   /* errno set by fork() */
   } else if (pid == 0) {                          /* child */
      if (*type == 'r') {
         close(pfd[0]);
         if (pfd[1] != STDOUT_FILENO) {
            dup2(pfd[1], STDOUT_FILENO);
            close(pfd[1]);
         }
      } else {
```

```
            close(pfd[1]);
            if (pfd[0] != STDIN_FILENO) {
                dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
            }
        }
        /* close all descriptors in childpid[] */
        for (i = 0; i < maxfd; i++)
            if (childpid[i] > 0)
                close(i);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);
    }
    /* parent continues... */
    if (*type == 'r') {
        close(pfd[1]);
        if ((fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
        close(pfd[0]);
        if ((fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }
    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}


int pclose(FILE *fp)
{
    int     fd, stat;
    pid_t   pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);     /* popen() has never been called */
    }

    fd = fileno(fp);
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);     /* fp wasn't opened by popen() */
    }

    childpid[fd] = 0;
```

```
    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */

    return(stat);   /* return child's termination status */
}
```

Program 7.3The popen and pclose functions

7.4. Co-processes

1. A UNIX system filter is a program that reads from standard input and writes to standard output.

2. Filters are normally connected linearly in shell pipelines. A filter becomes a co-process when the same program generates the filter's input and reads the filter's output.

3. The Korn shell provides co-processes. The Bourne shell, the Bourne-again shell, and the C shell don't provide a way to connect processes together as co-processes.

4. A co-process normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe.

5. Co-processes are also useful from a C program.

6. Whereas popen gives us a one-way pipe to the standard input or from the standard output of another process, with a co-process, we have two one-way pipes to the other process: one to its standard input and one from its standard output.

7. We want to write to its standard input, let it operate on the data, and then read from its standard output.

Example

8. Let's look at co-processes with an example. The process creates two pipes: one is the standard input of the co-process, and the other is the standard output of the co-process. Figure 7.7 shows this arrangement.

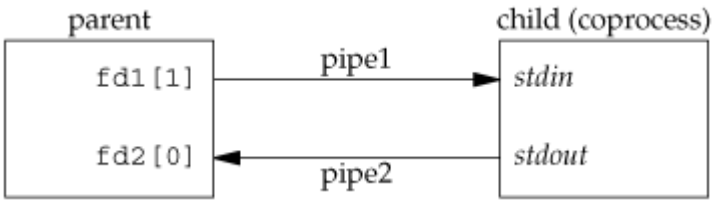


Figure 7.7. Driving a co-process by writing its standard input and reading its standard output

9. The program 7.4 is a simple co-process that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output. We compile this program and leave the executable in the file add2.

```
#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);
}
```

Program 7.4. Simple filter to add two numbers

The program 7.5 invokes the add2 co-process after reading two numbers from its standard input. The value from the co-process is written to its standard output.

```
#include "apue.h"

static void sig_pipe(int);     /* our signal handler */

int main(void)
{
   int    n, fd1[2], fd2[2];
   pid_t  pid;
   char   line[MAXLINE];

   if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
      err_sys("signal error");

   if (pipe(fd1) < 0 || pipe(fd2) < 0)
      err_sys("pipe error");
```

```
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                   /* parent */
        close(fd1[0]);
        close(fd2[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd1[1], line, n) != n)
                err_sys("write error to pipe");
            if ((n = read(fd2[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;    /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }
        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                                /* child */
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != STDIN_FILENO) {
            if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd1[0]);
        }
        if (fd2[1] != STDOUT_FILENO) {
            if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *)0) < 0)
            err_sys("execl error");
    }
    exit(0);
}
static void sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}
```

Program 7.5. Program to drive the add2 filter

10. Here, we create two pipes, with the parent and the child closing the ends they don't need.

11. We have to use two pipes: one for the standard input of the co-process and one for its standard output.

12. The child then calls dup2 to move the pipe descriptors onto its standard input and standard output, before calling execl.

13. If we compile and run the program 7.5, it works as expected. Furthermore, if we kill the add2 co-process while the program 7.5 is waiting for our input and then enter two numbers, the signal handler is invoked when the program writes to the pipe that has no reader.

7.5. FIFOs

1. FIFOs are sometimes called named pipes.

2. Pipes can be used only between related processes when a common ancestor has created the pipe.

3. With FIFOs, however, unrelated processes can exchange data.

4. We saw earlier that a FIFO is a type of file. One of the encodings of the st_mode member of the stat structure indicates that a file is a FIFO.

5. We can test for this with the S_ISFIFO macro.

6. Creating a FIFO is similar to creating a file. Indeed, the pathname for a FIFO exists in the file system.

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

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

Returns: 0 if OK, 1 on error

7. The specification of the mode argument for the mkfifo function is the same as for the open function.

8. The rules for the user and group ownership of the new FIFO are the same as we described earlier.

9. Once we have used mkfifo to create a FIFO, we open it using open.

10. Indeed, the normal file I/O functions (close, read, write, unlink, etc.) all work with FIFOs.

11. 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.

12. As with a pipe, if we write to a FIFO that no process has open for reading, the signal SIGPIPE is generated.

13. When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.

14. It is common to have multiple writers for a given FIFO. This means that we have to worry about atomic writes if we do not want the writes from multiple processes to be interleaved.

15. As with pipes, the constant PIPE_BUF specifies the maximum amount of data that can be written atomically to a FIFO.

16. There are two uses for FIFOs.

    1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.

    2. 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

17. FIFOs can be used to duplicate an output stream in a series of shell commands.

18. This prevents writing the data to an intermediate disk file (similar to using pipes to avoid intermediate disk files).

19. But whereas pipes can be used only for linear connections between processes, a FIFO has a name, so it can be used for nonlinear connections.

20. Consider a procedure that needs to process a filtered input stream twice. Figure 7.8 shows this arrangement.

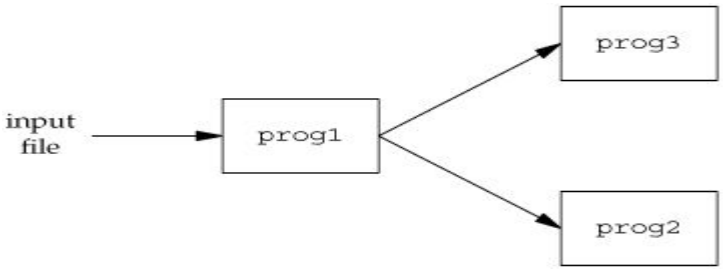


Figure 7.8. Procedure that processes a filtered input stream twice

21. 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
```

22. 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.

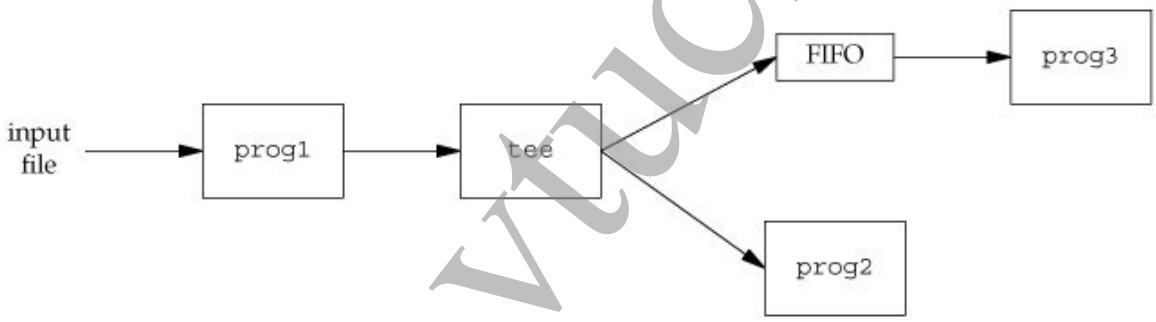


Figure 7.9 shows the process arrangement.

Figure 7.9. Using a FIFO and tee to send a stream to two different processes

ExampleClientServer Communication Using a FIFO

23. Another use for FIFOs is to send data between a client and a server.

24. 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. (By "well-known" we mean that the pathname of the FIFO is known to all the clients that need to contact the server.)

25. Figure 7.10 shows this arrangement.

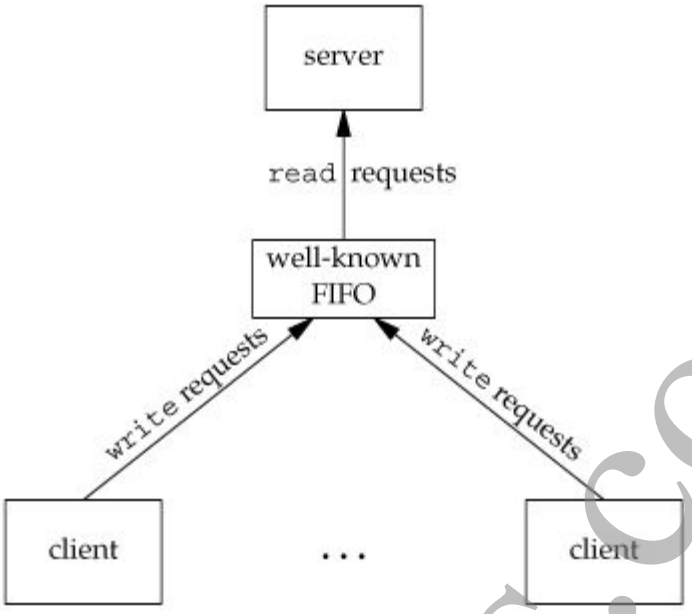


Figure 7.10. Clients sending requests to a server using a FIFO

26. 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.

27. The problem in using FIFOs for this type of client-server communication is how to send replies back from the server to each client.

28. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients.

29. 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's process ID.

30. For example, the server can create a FIFO with the name /tmp/serv1.XXXXX, where XXXXX is replaced with the client's process ID.

31. Figure 7.11 shows this arrangement.

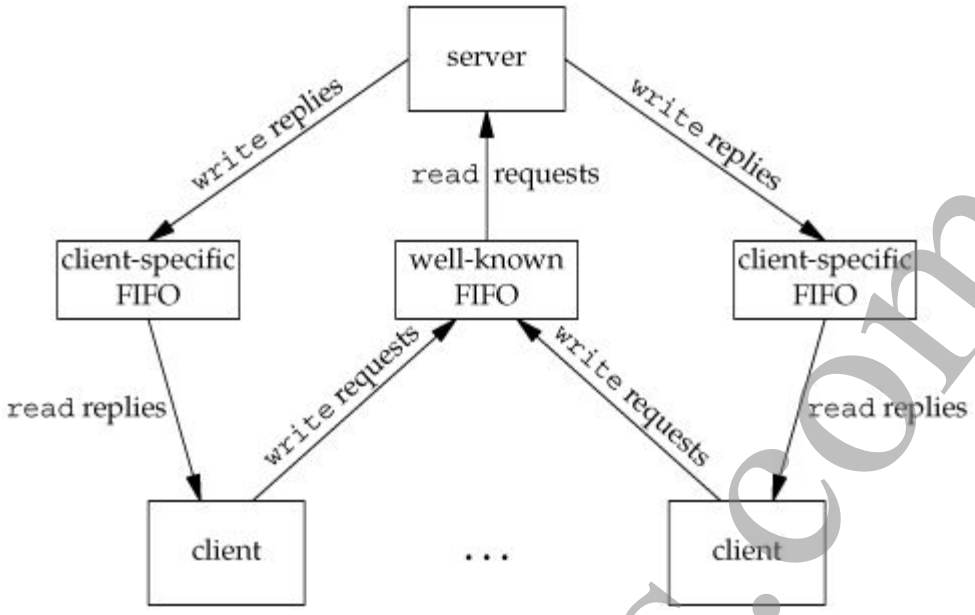


Figure 7.11. Clientserver communication using FIFOs

32. This arrangement works, although it is impossible for the server to tell whether a client crashes.

33. 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.

34. With the arrangement shown in Figure 7.11, if the server opens its well-known FIFO read-only (since it only reads from it) each time the number of clients goes from 1 to 0, the server will read an end of file on the FIFO. To prevent the server from having to handle this case, a common trick is just to have the server open its well-known FIFO for readwrite.

7.6. XSI- X/Open System Interface: IPC

1. The three types of IPC that are called XSI IPC are message queues, semaphores, and shared memory.

Identifiers and Keys

2. Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer identifier.

3. To send or fetch a message to or from a message queue, for example, all we need know is the identifier for the queue.

4. Unlike file descriptors, IPC identifiers are not small integers. Indeed, when a given IPC structure is created and then removed, the identifier associated with that structure continually increases until it reaches the maximum positive value for an integer, and then wraps around to 0.

5. The identifier is an internal name for an IPC object.

6. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object.

7. For this purpose, an IPC object is associated with a key that acts as an external name.

8. Whenever an IPC structure is being created (by calling msgget, semget, or shmget), a key must be specified.

9. 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>.

10. This key is converted into an identifier by the kernel.

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

    a. 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 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.

    b. 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.

    c. 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

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

13. 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.

14. If two pathnames refer to two different files, then ftok usually returns two different keys for the two pathnames.

15. 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.

16. The three get functions (msgget, semget, and shmget) all have two similar arguments: a key and an integer flag.

17. A new IPC structure is created (normally, by a server) if either key is IPC_PRIVATE or key is not currently associated with an IPC structure of the particular type and the IPC_CREAT bit of flag is specified.

18. To reference an existing queue (normally done by a client), key must equal the key that was specified when the queue was created, and IPC_CREAT must not be specified.

19. Note that it's never possible to specify IPC_PRIVATE to reference an existing queue, since this special key value always creates a new queue.

20. To reference an existing queue that was created with a key of IPC_PRIVATE, we must know the associated identifier and then use that identifier in the other IPC calls (such as msgsnd and msgrcv), bypassing the get function.

21. If we want to create a new IPC structure, making sure that we don't reference an existing one with the same identifier, we must specify a flag with both the IPC_CREAT and IPC_EXCL bits set. Doing this causes an error return of EEXIST if the IPC structure already exists.

Permission Structure

22. XSI IPC associates an ipc_perm structure with each IPC structure.

23. 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 */
  .
  .
  .
};
```

24. Each implementation includes additional members. See <sys/ipc.h> on your system for the complete definition.

25. All the fields are initialized when the IPC structure is created.

26. Later, we can modify the uid, gid, and mode fields by calling msgctl, semctl, or shmctl.

27. To change these values, the calling process must be either the creator of the IPC structure or the superuser.

28. Changing these fields is similar to calling chown or chmod for a file.

29. The values in the mode field are as shown below.

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

30. Below are the six permissions for each form of IPC.

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

Configuration Limits

31. All three forms of XSI IPC have built-in limits that we may encounter.

32. Most of these limits can be changed by reconfiguring the kernel.

Advantages and Disadvantages

33. A fundamental problem with XSI IPC is that the IPC structures are system wide and do not have a reference count.

34. 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.

35. They remain in the system until specifically read or deleted by some process calling msgrcv or msgctl, by someone executing the ipcrm command, or by the system being rebooted.

36. Another problem with XSI IPC is that these IPC structures are not known by names in the file system.

37. We can't access them and modify their properties.

38. Almost a dozen new system calls (msgget, semop, shmat, and so on) were added to the kernel to support these IPC objects.

39. 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.

40. Instead, two new commands ipcs(1) and ipcrm(1)were added.

41. Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (select and poll) with them.

42. 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.

43. Other advantages for message queues are that they're reliable, flow controlled, record oriented, and can be processed in other than first-in, first-out order.

44. The following compares some of the features of these various forms of IPC.

IPC type	Connectionless?	Reliable?	Flow control?	Records?	Message types or priorities?
message queues	no	yes	yes	yes	yes
STREAMS	no	yes	yes	yes	yes
UNIX domain stream socket	no	yes	yes	no	no
UNIX domain datagram socket	yes	yes	no	yes	no
FIFOs (non-STREAMS)	no	yes	yes	no	no

7.7. Message Queues

1. A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.

2. We will call the message queue just a queue and its identifier a queue ID.

3. A new queue is created or an existing queue opened by msgget.

4. New messages are added to the end of a queue by msgsnd.

5. 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.

6. Messages are fetched from a queue by msgrcv.
7. We do not have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

8. Each queue has the following msqid_ds structure associated with it:

```
struct msqid_ds {
    struct ipc_perm     msg_perm;
    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 */
    .
    .
    .
};
```

9. This structure defines the current status of the queue. The members shown are the ones defined by the Single UNIX Specification. Implementations include additional fields not covered by the standard.

10. 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
```

11. We have already seen the rules for converting the key into an identifier and discussed whether a new queue is created or an existing queue is referenced.

12. 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.

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

14. 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

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

IPC_STAT	Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf.
IPC_SET	Copy the following fields from the structure pointed to by buf to the msqid_ds structure associated with this queue: msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes. This command can be executed only by a process whose effective user ID equals msg_perm.cuid or msg_perm.uid or by a process with superuser privileges. Only the superuser can increase the value of msg_qbytes.
IPC_RMID	Remove the message queue from the system and any data still on the queue. This removal is immediate. Any other process still using the message queue will get an error of EIDRM on its next attempted operation on the queue. This command can be executed only by a process whose effective user ID equals msg_perm.cuid or msg_perm.uid or by a process with superuser privileges.

16. 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

17. As we mentioned earlier, 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).

18. Messages are always placed at the end of the queue.

19. 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.)

20. 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 */
  };
```

21. 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.

22. A flag value of IPC_NOWAIT can be specified. This is similar to the nonblocking I/O flag for file I/O.

23. If the message queue is full, specifying IPC_NOWAIT causes msgsnd to return immediately with an error of EAGAIN.

24. If IPC_NOWAIT is not specified, we are blocked until there is room for the message, the queue is removed from the system, or a signal is caught and the signal handler returns.

25. In the second case, an error of EIDRM is returned ("identifier removed"); in the last case, the error returned is EINTR.

26. Note how ungracefully the removal of a message queue is handled. Since a reference count is not maintained with each message queue, the removal of a queue simply generates errors on the next queue operation by processes still using the queue.

27. When msgsnd returns successfully, the msqid_ds structure associated with the message queue is updated to indicate the process ID that made the call (msg_lspid), the time that the call was made (msg_stime), and that one more message is on the queue (msg_qnum).

28. 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

29. The ptr argument points to a long integer (where the message type of the returned message is stored) followed by a data buffer for the actual message data.

30. nbytes specifies the size of the data buffer. If the returned message is larger than nbytes and the MSG_NOERROR bit in flag is set, the message is truncated.

31. 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.

32. A nonzero type is used to read the messages in an order other than first in, first out.

33. For example, the type could be a priority value if the application assigns priorities to the messages. Another use of this field is to contain the process ID of the client if a single message queue is being used by multiple clients and a single server (as long as a process ID fits in a long integer).

34. We can specify a flag value of IPC_NOWAIT to make the operation nonblocking, causing msgrcv to return -1 with errno set to ENOMSG if a message of the specified type is not available.

35. If IPC_NOWAIT is not specified, the operation blocks until a message of the specified type is available, the queue is removed from the system (-1 is returned with errno set to EIDRM), or a signal is caught and the signal handler returns (causing msgrcv to return 1 with errno set to EINTR).

36. When msgrcv succeeds, the kernel updates the msqid_ds structure associated with the message queue to indicate the caller's process ID (msg_lrpid), the time of the call (msg_rtime), and that one less message is on the queue (msg_qnum).

7.8. Semaphores

1. A semaphore is not a form of IPC. A semaphore is a counter used to provide access to a shared data object for multiple processes.

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

   a.) Test the semaphore that controls the resource.

   b.) 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.

   c.) 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.

3. 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.

4. To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel.

5. A common form of semaphore is called a binary semaphore. It controls a single resource, and its value is initialized to 1.

6. 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.

7. XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

   a.) 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.

   b.) 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.

   c.) 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.

8. The kernel maintains a semid_ds structure for each semaphore set:

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

9. The Single UNIX Specification defines the fields shown, but implementations can define additional members in the semid_ds structure.

10. 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 */
    .
    .
    .
};
```

11. The table below lists the system limits that affect semaphore sets.

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
The maximum value of any semaphore	32,767	32,767	32,767	32,767
The maximum value of any semaphore's adjust-on-exit value	16,384	32,767	16,384	16,384
The maximum number of semaphore sets, systemwide	10	128	87,381	10
The maximum number of semaphores, systemwide	60	32,000	87,381	60
The maximum number of semaphores per semaphore set	60	250	87,381	25
The maximum number of undo structures, systemwide	30	32,000	87,381	30
The maximum number of undo entries per undo structures	10	32	10	10
The maximum number of operations per semop call	100	32	100	10

12. 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

13. We have already seen the rules for converting the key into an identifier and discussed whether a new set is created or an existing set is referenced.

14. 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.

15. The number of semaphores in the set is nsems.

16. 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.

17. 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 */);
```

18. 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 */
};
```

19. Note that the optional argument is the actual union, not a pointer to the union.

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

21. The five commands that refer to one particular semaphore value use semnum to specify one member of the set. The value of semnum is between 0 and nsems-1, inclusive.

IPC_STAT	Fetch the semid_ds structure for this set, storing it in the structure pointed to by arg.buf.

IPC_STAT	Fetch the semid_ds structure for this set, storing it in the structure pointed to by arg.buf.
IPC_SET	Set the sem_perm.uid, sem_perm.gid, and sem_perm.mode fields from the structure pointed to by arg.buf in the semid_ds structure associated with this set. This command can be executed only by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid or by a process with superuser privileges.
IPC_RMID	Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of EIDRM on its next attempted operation on the semaphore. This command can be executed only by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid or by a process with superuser privileges.
GETVAL	Return the value of semval for the member semnum.
SETVAL	Set the value of semval for the member semnum. The value is specified by arg.val.
GETPID	Return the value of sempid for the member semnum.
GETNCNT	Return the value of semncnt for the member semnum.
GETZCNT	Return the value of semzcnt for the member semnum.
GETALL	Fetch all the semaphore values in the set. These values are stored in the array pointed to by arg.array.
SETALL	Set all the semaphore values in the set to the values pointed to by arg.array.

22. For all the GET commands other than GETALL, the function returns the corresponding value. For the remaining commands, the return value is 0.

23. 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

24. 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
*/
};
```

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

26. The operation on each member of the set is specified by the corresponding sem_op value.

27. This value can be negative, 0, or positive.

a.) The easiest case is when sem_op is positive. This case corresponds to the returning of resources by the process. The value of sem_op is added to the semaphore's value. If the undo flag is specified, sem_op is also subtracted from the semaphore's adjustment value for this process.

b.) If sem_op is negative, we want to obtain resources that the semaphore controls. If the semaphore's value is greater than or equal to the absolute value of sem_op (the resources are available), the absolute value of sem_op is subtracted from the semaphore's value. This guarantees that the resulting value for the semaphore is greater than or equal to 0. If the undo flag is specified, the absolute value of sem_op is also added to the semaphore's adjustment value for this process.

If the semaphore's value is less than the absolute value of sem_op (the resources are not available), the following conditions apply.

a. If IPC_NOWAIT is specified, semop returns with an error of EAGAIN.
b. If IPC_NOWAIT is not specified, the semncnt value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.
    i. The semaphore's value becomes greater than or equal to the absolute value of sem_op (i.e., some other process has released some resources). The value of semncnt for this semaphore is decremented (since the calling process is done waiting), and the absolute value of sem_op is subtracted from the semaphore's value. If the undo flag is specified, the absolute value of sem_op is also added to the semaphore's adjustment value for this process.
    ii. The semaphore is removed from the system. In this case, the function returns an error of EIDRM.
    iii. A signal is caught by the process, and the signal handler returns. In this case, the value of semncnt for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of EINTR.

c.) If sem_op is 0, this means that the calling process wants to wait until the semaphore's value becomes 0.

If the semaphore's value is currently 0, the function returns immediately.
If the semaphore's value is nonzero, the following conditions apply.

a. If IPC_NOWAIT is specified, return is made with an error of EAGAIN.
b. If IPC_NOWAIT is not specified, the semzcnt value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.
    i. The semaphore's value becomes 0. The value of semzcnt for this semaphore is decremented (since the calling process is done waiting).
    ii. The semaphore is removed from the system. In this case, the function returns an error of EIDRM.
    iii. A signal is caught by the process, and the signal handler returns. In this case, the value of semzcnt for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of EINTR.

28. The semop function operates atomically; it does either all the operations in the array or none of them.

Semaphore Adjustment on exit

29. It is a problem if a process terminates while it has resources allocated through a semaphore.

30. Whenever we specify the SEM_UNDO flag for a semaphore operation and we allocate resources (a sem_op value less than 0), the kernel remembers how many resources we allocated from that particular semaphore (the absolute value of sem_op).

31. When the process terminates, either voluntarily or involuntarily, the kernel checks whether the process has any outstanding semaphore adjustments and, if so, applies the adjustment to the corresponding semaphore.

32. If we set the value of a semaphore using semctl, with either the SETVAL or SETALL commands, the adjustment value for that semaphore in all processes is set to 0.

UNIT - 8

NETWORK IPC: SOCKETS

Socket Descriptors

- **A socket is an abstraction of a communication endpoint.**
- **To create a socket we can make a call the following function.**

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

int socket(int domain, int type, int protocol);

                                    Returns: file (socket) descriptor if OK, 1 on error
```

- **The socket communication domains**

Domain	Description
AF_INET	IPv4 Internet domain
AF_INET6	IPv6 Internet domain
AF_UNIX	UNIX domain
AF_UNSPEC	unspecified

Type	Description
SOCK_DGRAM	fixed-length, connectionless, unreliable messages
SOCK_RAW	datagram interface to IP (optional in POSIX.1)
SOCK_SEQPACKET	fixed-length, sequenced, reliable, connection-oriented messages
SOCK_STREAM	sequenced, reliable, bidirectional, connection-oriented byte streams

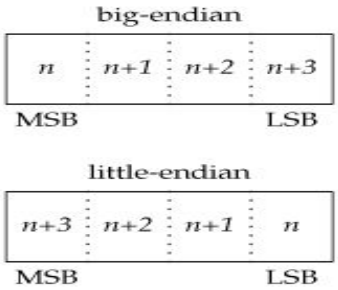
```
#include <sys/socket.h>

int shutdown (int sockfd, int how);

                                    Returns: 0 if OK, 1 on error
```

Addressing

- **Byte Ordering**



- **Byte order for test platforms**

Operating system	Processor architecture	Byte order
FreeBSD 5.2.1	Intel Pentium	little-endian
Linux 2.4.22	Intel Pentium	little-endian
Mac OS X 10.3	PowerPC	big-endian
Solaris 9	Sun SPARC	big-endian

Functions to convert between network byte order & Processor byte order

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostint32);

                                Returns: 32-bit integer in network byte order
uint16_t htons(uint16_t hostint16);

                                Returns: 16-bit integer in network byte order
uint32_t ntohl(uint32_t netint32);

                                Returns: 32-bit integer in host byte order
uint16_t ntohs(uint16_t netint16);

                                Returns: 16-bit integer in host byte order
```

Address Formats

- struct sockaddr
{
sa_family_t sa_family; /* address family */
char sa_data[]; /* variable-length address */ . .
.
};

- struct sockaddr
{
sa_family_t sa_family; /* address family */
char sa_data[14]; /* variable-length address */
};
struct in_addr
{
in_addr_t s_addr; /* IPv4 address */
};

struct sockaddr_in
{
sa_family_t sin_family; /* address family */
in_port_t sin_port; /* port number */
struct in_addr sin_addr; /* IPv4 address */
};

struct in6_addr
{
uint8_t s6_addr[16]; /* IPv6 address */
};

struct sockaddr_in6
{
sa_family_t sin6_family; /* address family */
in_port_t sin6_port; /* port number */
uint32_t sin6_flowinfo; /* traffic class and flow info */
struct in6_addr sin6_addr; /* IPv6 address */
uint32_t sin6_scope_id; /* set of interfaces for scope */
};

To print the address in Human understandable form

[View full width]

```
#include <arpa/inet.h>

const char *inet_ntop(int domain, const void
    *restrict addr,
                      char *restrict str,
    socklen_t size);
```

Returns: pointer to address string on success, NULL on error

```
int inet_pton(int domain, const char *restrict str,
              void *restrict addr);
```

Returns: 1 on success, 0 if the format is invalid, or 1 on error

Address Lookup

- The hosts known by a given computer system

```
#include <netdb.h>

struct hostent *gethostent(void);
```

Returns: pointer if OK, NULL on error

```
void sethostent(int stayopen);

void endhostent(void);
```

```
struct hostent
{
    char   *h_name;      /* name of host */
    char  **h_aliases;   /* pointer to alternate host name array */
    int     h_addrtype;  /* address type */
    int     h_length;    /* length in bytes of address */
    char  **h_addr_list; /* pointer to array of network addresses */
    .
    .
    .
  };
```

network names and numbers with a similar set of interfaces

```
#include <netdb.h>

struct netent *getnetbyaddr(uint32_t net, int type);

struct netent *getnetbyname(const char *name);

struct netent *getnetent(void);

                                    All return: pointer if OK, NULL on error

void setnetent(int stayopen);

void endnetent(void);
```

struct netent
{
    char   *n_name;   /* network name */
    char   **n_aliases;   /* alternate network name array pointer */
    int   n_addrtype; /* address type */
    uint32_t n_net;   /* network number */
     };

To map between protocol names and numbers

```
#include <netdb.h>

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotoent(void);

                                    All return: pointer if OK, NULL on error

void setprotoent(int stayopen);

void endprotoent(void);
```

struct protoent
{

```
    char   *p_name;     /* protocol name */
    char  **p_aliases;  /* pointer to alternate protocol name array */
    int     p_proto;    /* protocol number */
    .
    .
    .
};
```

To Map between Service name to port number and vice versa

[View full width]

```
#include <netdb.h>

struct servent *getservbyname(const char *name,
➥  const char *proto);

struct servent *getservbyport(int port, const char
➥  *proto);

struct servent *getservent(void);
```

All return: pointer if OK, NULL on error

```
void setservent(int stayopen);

void endservent(void);
```

```
struct servent
{
    char   *s_name;     /* service name */
    char  **s_aliases;  /* pointer to alternate service name array */
    int     s_port;     /* port number */
    char   *s_proto;    /* name of protocol */
    .
    .
    .
};
```

To map from a host name and a service name to an address and vice versa

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict host,
                const char *restrict service,
                const struct addrinfo *restrict hint,
                struct addrinfo **restrict res);


                                Returns: 0 if OK, nonzero error code on error
void freeaddrinfo(struct addrinfo *ai);
```

```
struct addrinfo {
    int             ai_flags;       /* customize behavior */
    int             ai_family;      /* address family */
    int             ai_socktype;    /* socket type */
    int             ai_protocol;    /* protocol */
    socklen_t       ai_addrlen;     /* length in bytes of address */
    struct sockaddr  *ai_addr;      /* address */
    char            *ai_canonname;  /* canonical name of host */
    struct addrinfo  *ai_next;      /* next in list */
    .   };
```

Flags for addrinfo structure

Flag	Description
AI_ADDRCONFIG	Query for whichever address type (IPv4 or IPv6) is configured.
AI_ALL	Look for both IPv4 and IPv6 addresses (used only with AI_V4MAPPED).
AI_CANONNAME	Request a canonical name (as opposed to an alias).
AI_NUMERICHOST	Return the host address in numeric format.
AI_NUMERICSERV	Return the service as a port number.
AI_PASSIVE	Socket address is intended to be bound for listening.
AI_V4MAPPED	If no IPv6 addresses are found, return IPv4 addresses mapped in IPv6 format.

To Handle the error Messages

```
#include <netdb.h>
const char *gai_strerror(int error);
```

To converts an address into a host name and a service name

[View full width]

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *restrict addr,
                socklen_t alen, char *restrict host,
                socklen_t hostlen, char *restrict
➥ service,
                socklen_t servlen, unsigned int
➥ flags);
```

Returns: 0 if OK, nonzero on error

Associating Addresses with Sockets

[View full width]

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

int getpeername(int sockfd, struct sockaddr
➥ *restrict addr,
                socklen_t *restrict alenp);
```

Returns: 0 if OK, 1 on error

[View full width]

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

int bind(int sockfd, const struct sockaddr *addr,
➥ socklen_t len);
```

Returns: 0 if OK, 1 on error