UNIT-1

INTRODUCTION

1 .What is posix standard? Explain the different subset of posix standard. Write structure of program to filter out non-posix compliant codes from user program.

(06 marks) (DEC-2010).

Ans: The IEEE society formed a special task force called POSIX. To create a set of standards for operation systems interfaceing.several sub roots are POSIX.1,POSIX.1b AND POSIX.1c

POSIX.1 committee proposes is standard for base OS API this is standard specifies APIs for manipulating files and process. It is known as IEEE 1003.1-1990.the POSIX.1B committee proposes a set of standard APIs for RTOS interface these include IPC. This is known as IEEE standard 1003.4-1993.the POSIX.1c standard specifies multi threaded programming interface.

```
#define   _POSIX_SOURCE
#define   _POSIX_C_SOURCE                199309L
#include<iostream.h>
#include<unistd.h>
Int main()
{
#ifdef  _POSIX_VERSION
Cout<<"System conforms to posix"<<_POSIX_VERSION<<endl;
#else
Cout<<"_POSIX_VERSION is undefined\n";
#endif
Return 0;
}
```

2.Write a program c or c++ posix complement program to check following limits:
i)number of clock ticks
ii)Maximum number of child processes
iii)Maximum path length
iv)Maximum number of characters in file name
v)Maximum number of open files per process.   (08 marks) Dec 2009/May 2012
Ans:

```
#define   _POSIX_SOURCE
```

```
#define   _POSIX_C_SOURCE                  199309L
#include<iostream.h>
#include<unistd.h>
Int main()
{
  Int res;
If((res=sysconf(_SC_OPEN_MAX)==-1)
perror(“sysconf”);
else cout<<”OPEN_MAX:”<<res<<endl;

if((res=pathconf(“/”,_PC_NAME_MAX)==-1)
perror(“pathconf”);
else
cout<<”max_path name:<<(res+1)<<endl;

If((res=sysconif(_SC_CLK_PCK)==-1)
perror(“sysconf”);
else cout<<”clock ticks:”<<res<<endl;

if((res= sysconf (“/”,_SC_CHILD_MAX)==-1)
perror(“sysconf”);
else
cout<<”max_childs:”<<(res+1)<<endl;

if((res=pathconf(“/”,_PC_PATH_MAX)==-1)
perror(“pathconf”);
else
cout<<”max path name length:<<(res+1)<<endl;
return 0;
}
```

3.Explain the common characteristics of API and describe the error status codes

(06 marks) May 2009/May 2012

Ans:

The APIs return an integer value it indicate the termination status of their execution.if an API returns value -1 it means API failed and the global variable errno is set with an error code.user processes main call the peroor function to print diagnostic message of failure to standard output or it may call strerror function and gives it errno as actuvall argument value, the strerror function returns message string and the user process to print that message.

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

4. Bring out the importance of standardizing the UNIX OS. What aspects of c programming language have been standardized in ANSI c? With suitable example bring out the two important difference b/n K&R C and ANSIC with respect to function prototyping and pointers to function. (08 marks) Dec-09/May -12.

Ans:

The IEEE society formed a special task force called POSIX. To create a set of standards for operation systems interfacing. Several sub roots are POSIX.1,POSIX.1b AND POSIX.1c

POSIX.1 committee proposes is standard for base OS API this is standard specifies APIs for manipulating files and process. It is known as IEEE 1003.1-1990.the POSIX.1B committee proposes a set of standard APIs for RTOS interface these include IPC. This is known as IEEE standard 1003.4-1993.the POSIX.1c standard specifies multi threaded programming interface.

The major aspect are:

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

ANSI C adopts function prototyping where function definition and declaration include name,argument data type and return value data type. This enables comliersto check for function calls in user programs that pass invlaid no. of arguments or incompatible argument data type.

For example

Unsigned long foo(char* fmt,double data);

ANSI C specifies that function pointer may be used like function name. No dereferenceis needed when calling a function whose address is contained in a pointer. Example the following statement define a function pointer funptr which contains the address of function foo:

Extern void foo(double xyz,const int* lptr);

Void (*funcptr)(double,const int*)=foo;

5. Write a c++ program to list the actual values of the following system configuration limits on a given unix OS.
i) maximum no. of child processes that can be created.
ii) maximum no. of files that can be opened simultaneously.
ii)maximum no. of message queues that can be accessed. (06 marks) Dec-10/May-12.
Ans:

```
#define  _POSIX_SOURCE
#define  _POSIX_C_SOURCE                199309L
#include<iostream.h>
#include<unistd.h>
Int main()
{
  Int res;
If((res=sysconf(_SC_OPEN_MAX)==-1)
perror("sysconf");
else cout<<"OPEN_MAX:"<<res<<endl;

if((res=pathconf("/",_PC_NAME_MAX)==-1)
perror("pathconf");
else
```

```
cout<<"max_path name:<<(res+1)<<endl;

If((res=sysconif(_SC_CLK_PCK)==-1)
perror("sysconf");
else cout<<"clock ticks:"<<res<<endl;

if((res= sysconf ("/",_SC_CHILD_MAX)==-1)
perror("sysconf");
else
cout<<"max_childs:"<<(res+1)<<endl;

if((res=pathconf("/",_PC_PATH_MAX)==-1)
perror("pathconf");
else
cout<<"max path name length:<<(res+1)<<endl;
return 0;

}
```

6. List the differences between ANSI C and K & R C.Explain (05 marks)

May-06/Dec-06/May-08/Dec-09.

Ans:

The major difference b/n ANSI C and K&R C are:

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

ANSI C adopts function prototyping where function definition and declaration include name,argument data type and return value data type. This enables comliersto check for function calls in user programs that pass invlaid no. of arguments or incompatible argument data type.

For example

Unsigned long foo(char* fmt,double data);

ANSI C specifies that function pointer may be used like function name. No dereferenceis needed when calling a function whose address is contained in a pointer. Example the following statement define a function pointer funptr which contains the address of function foo:

Extern void foo(double xyz,const int* lptr);

Void (*funcptr)(double,const int*)=foo;

7. Explain any five error status code for error no. (05 marks) Dec-10.

Ans:

The APIs return an integer value it indicate the termination status of their execution.if an API returns value -1 it means API failed and the global variable errno is set with an error code.user processes main call the peroor function to print diagnostic message of failure to standard output or it may call strerror function and gives it errno as actuvall argument value, the strerror function returns message string and the user process to print that message.

Error status code	meaning
EACCESS	a process does not have acess permission to perform an operation via a API.
EPERM	a API was abotrted because the calling process does’t have the superuser privilege
ENOENT	an invalid file name was specified to an API
EAGAIN	a API was aborted because some system resource it required was temporarily unavailable.The API should be called again later.
ENOMEM	a API was aborted because it could not allocate dyanamic memory
EIO	I/O error occurred in an API execution

UNIT - 2

UNIX FILES

1. Explain the different file types available in UNIX or POSIX system. (08 marks)

(May-08/Dec-08/May-10/May-12)

Ans:

Regular File: This may be either text or binary file and may be executable provided the execution rights of the file are set and this are written by user,can be created using vi editor.

It is very flexible to create, read, write ,modify and removed by specific system commands.

Directory file: It is like a file folder that contains another files, including subdirectory files. It has two fields filename and inode number. A directory may be created using mkdir command as mkdir /usr/suman the directory file is considered to be empty if it contains no other file except "." And ".." files.

FIFO file: It is a special pipe device file which provides a temporary buffer for two or more process to communicate by writing data to and reading data from buffer. The size of the buffer associated with FIFO file is fixed to PIE_BUF. Once the IPC begins then only memory will be allocated for FIFO file and it becomes physically existing. During IPC the data written to FIFO may exceed the PIPE_BUF then writer process will be blocked until a reader make a read. It can be created using

mkfifo /dev/fifo5

Character device file: It represents the physical device that transmits data in a character based manner. Example line printer, console. For a hard disk these are used to raw data transfer between process and disk. OS will automatically invoke appropriate device driver function to perform the data transfer.

Block device file: It represents a physical device that transmits data block at a time.Example floppy disk drives.

It can be created by

```
mknod  /dev/cdk1  c 115 5           for character device file

mknod  /dev/bdk1  b 150 15           for block device file
```

second argument to the command is ‘c’ for character device file and ‘b’ for block device file. Other arguments represent major and minor number. A major number is an index of the kernel table that contains the address of device driver functions known to the system. One device driver can serve many devices and they require minor number to serve for the deivces. Minor number is passed as argument to device driver when it is called.

2. What is an API? How is it different from c library functions? Why calling an API is more time consuming then calling an user function. (06 marks) (May-07/Dec-08)

Ans:

Unix provides a set off applications programming interface which may be called by user program to perform system specific functions. These functions allow users application to directly manipulate system objects such as files and process that can be done by using c library functions. Furthermore c library functions call these APIs to perform to actual worked advantage thus users may use theses APIs directly to by pass over head of calling a c library functions.

Most APIs access kernel internal resources. Thus when an APIs invoke by a process the execution context of the process is switch by the kernel from user mode to kernel mode. When an APIs execution completes a user process is switched back to user mode. This context switching for API call ensures that process access kernel data in controlled manner and minimizes any chance of runaway application damage in the system. Hence API is more time consuming.

3. Describe the UNIX Kernel support for files. (06 marks) (May-07/Dec-08/May-12)

Ans:

Kernel has the file table that keep track of all opened files in the system. There is also inode table that contains the different attributes of the files recently accesed.

When a user excutes a command, a process is created by the kernel to carry out the command execution and the process has its own data structure and it has an file descriptor table.FDT has a OPEN_MAX entries and record all the files opened by the process. When open function is called the kernel will resolove the pathnme to the file inode.

If the file inode is not found or the process lacks appropriate permission to acess the inode data **,** If open call fails then it returns -1.

The process for this are as follows:

- The kernel will search the process FDT and look for the first unused entry.If found, that entry will be designed to refence file.
  The index to the entry will be returned to the processes as the file descriptor of the opened file.
- `the kernel will scan the file table in its kernel space to find an a unused entry that can be assigned to the reference file.
  If an unused entry is found,then the following events will occur.
  a)the processes FDT entry will be set to point to this file table entry.
  b)the FT entry will contain the current file pointer of the open file.and this is an a offset from the beginning of the file.where read (or) write as to be done.
  c) the FT entry will be set to point to the inode table entry where the inode record of the file is stored.

d) FT entry contain an open mode that the file is opened for read-only,write-only.call for both.the openmode specified from the calling process as an arrugument to open function call.

e) The reference count in the file table entry is set to 1.and it keep track of how many file descriptor from any processes are referring the entry.

f) the reference count of the in –memory inode of the file is increased by 1.

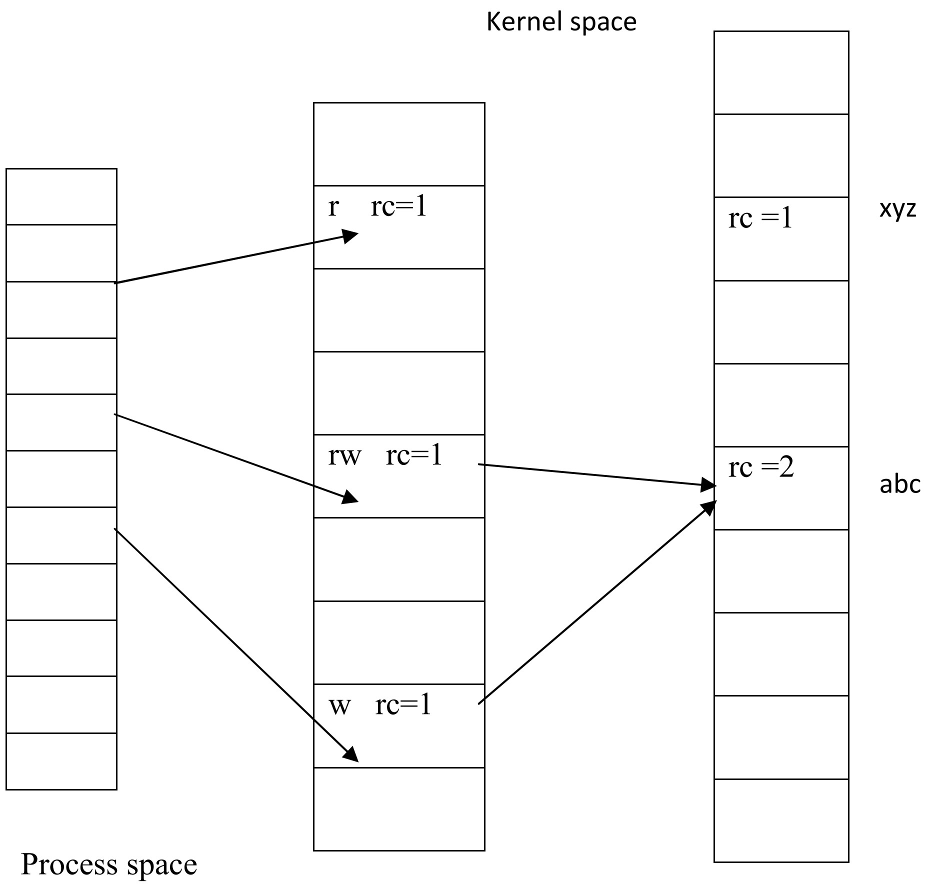
The fig. shows the FDT, the kernel file table and the inode table after the process as opened three files x,y,z for read only,abc for write/read ,and abc for write only.the reference count in a file inode record specifies how many file table,entres are pointing to the file inode record.if the count is not 0,it means that 1 or more process are currently opening the file for access.

If the process call the seek system call to change file pointer to a different offset for next read operation,the kernel will use the file descriptor to the index.the process FDT to find the pointer to the table entry then kernel access this to get pointer point to file inode record and then it check for a file.if the file type is competable with seek,the kernel will change the file pointer in the FT according to the value specified in the lseek argument.

When the process calls a close function to the close an open file, the sequence of events are as bellow.

- The kernel sets the corresponding FDT entry to be unused.
- It decrements the reference count in the correspanding FDT entry by 1.if the reference count is still non-zero then goto last step.
- The file table entry is marked unused.
- RC in the corrsepanding file inode table entry is decremented by 1.if the count still non-zero it goto last step.
- If the hard link count of the inode is not zero it returns the caller with success status.otherwise inode table entry will made unused and deallocates all physical disk storage of the file,as all the path names have been removed by some process.
- It returns to the process with 0 on success status.

File descriptor table file table inode table



Rc=reference count

R=read only

W=write only

RW read write

4. What are APIs? When do you use them? Why are the APIs more time consuming than the library function? (03 marks) (May-06/Dec-06/May-07/Dec-08)

Ans:

Unix provides a set off all applications programming interface which may be called by user program to perform system specific functions. These functions allow users application to directly manipulate system objects such as files and process that can be done by using c library functions. Furthermore c library functions call these APIs to perform to actual worked advantage thus users may use theses APIs directly to by pass over head of calling a c library functions.

Most APIs access kernel internal resources. Thus when an APIs invoke by a process the execution context of the process is switch by the kernel from user mode to kernel mode. When an APIs execution completes a user process is switched back to user mode. This context switching for API call ensures that process access kernel data in controlled manner and minimizes any chance of runaway application damage in the system. Hence API is more time consuming.

5. What are the API common characteristics?.List any four values of the global variable errno along with their meaning wherever the APIs fail.(05 marks) (May-05)

Ans:

The APIs return an integer value it indicate the termination status of their execution.if an API returns value -1 it means API failed and the global variable errno is set with an error code.user processes main call the peroor function to print diagnostic message of failure to standard output or it may call strerror function and gives it errno as actuvall argument value, the strerror function returns message string and the user process to print that message.

Error status code	meaning
EACCESS	a process does not have acess permission to perform an operation via a API.

EPERM	a API was abotrted because the calling process does't have the superuser privilege
ENOENT	an invalid file name was specified to an API
BADF	A Api was called with an invalid file descriptor
EINTR	a API execution was aborted due to a signal interruption

6. List all the file attributes along with their meanings. Which of these attributes can't be changed and why? List the commands needed to change the following file attributes. i) file size; ii) User ID; iii) Last access and modification time; iv) hard link count. (05marks) (May-06/Dec-08)

Ans:

Attribute	value meaning
File type	type of file
Access permission	the file access permission for owner,group and others
Hard link count	no. of hard links of a file
UID	the file owner user ID
GID	the file group ID
File size	the file size in bytes
Last access time	the time the file was last accessed
Last modified time	the time the file was last modified
Last change time	the time the file access permission, UID,GID or hard link count was last changed
Inode number	the system inode number of the file
File system Id	the file system id where the file is stored

The following attributes can't be changed as they stay unchanged for the entire life of the file

File type,file inode number,file system id ,major and minor device number.

Commands to change attribute are

Attribute	command
File size	vi, emac
User Id	chown
Last access & modification time	touch,vi,emac
Hard link count	ln

7. What is an inode? Why are the inodes unique only within a file system? How does OS map the inode to its filename? Bring out the four important differences between soft and hard links. (07 marks) (May-08/Dec-09/May-12)

Ans:

Inode is an attribute of file,it is an unique no. used refer an inode table.

An operating system may have access to multiple file system at one timeand an inode number is unique with in a file sytem only, a file inode record is identified by a file system id and an inode number.

An OS does not keep the name of a file in it's inode record,because the mapping of file names to inode no.s is done via directory file.Specifically a directory file conteains a list of names and their respective inode numbers for all files stored in that directory.for example, if a directory foo contains files xyz,a.out and xyz_ln1,where xyz_ln1 is a hard link of xyz. To acess a file say xyz, the directory file is searched with either inode no. or file name.Hence mapping is done.

Hard link	soft link
Does not create a new inode	create a new inode number
Can't link directories,unless done by root	Can link directories
Can't link file s across the file systems	Can link file s across the file systems
Increase the hard link count of the linked inode	does't change the hard link count of the linked inode

8. What are different categories of file types available with UNIX? Explain. Give some commands to create each of them. (10 marks) (May-10/Dec-11)

Ans:

Regular File: This may be either text or binary file and may be executable provided the execution rights of the file are set and this are written by user,can be created using vi editor.

It is very flexible to create,read,write ,modify and removed by specific system commands.

Directory file: It is like a file folder that contains another files,including subdirectory files. It has two fields filename and inode number. A directory may be created using mkdir command as

mkdir /usr/suman

the directory file is considered to be empty if it contains no other file except "." And ".." files.

FIFO file: It is a special pipe device file which provides a temporary buffer for two or more process to communicate by writing data to and reading data from buffer. The size of the buffer associated with FIFO file is fixed to PIE_BUF. Once the IPC begins then only memory will be allocated for FIFO file and it becomes physically existing. During IPC the data written to FIFO may exceed the PIPE_BUF then writer process will be blocked until a reader make a read.It can be created using

mkfifo /dev/fifo5

Character device file: It represents the physical device that transmits data in a character based manner. Example line printer, console. For a hard disk these are used to raw data transfer between process and disk. OS will automatically invoke appropriate device driver function to perform the data transfer.

Block device file: It represents a physical device that transmits data block at a time.Example floppy disk drives.

It can be created by

```
mknod  /dev/cdk1  c 115 5           for character device file

mknod  /dev/bdk1  b 150 15           for block device file
```

second argument to the command is ‘c’ for character device file and ‘b’ for block device file. Other arguments represent major and minor number. A major number is an index of the kernel table that contains the address of device driver functions known to the system. One device driver can serve many devices and they require minor number to serve for the deivces. Minor number is passed as argument to device driver when it is called

UNIT - 3

UNIX File APIs

1. **List and explain the access mode flags and access modifier flags. Also explain how the permission value specified in an 'open' call is modified by its calling process 'umask' value. (04 marks) (May-08/May-09/Dec-10/May-12)**

Ans:

Access mode flag	**use**
O_RDONLY	opens the file for read only
O_WRONLY	opens the file for write only
O_RDWR	opens the file for read and write
O_APPEND	appends data to the end of the file
O_CREAT	creates the file if it does't exists
O_EXCL	used with the O_CREAT only. This flag causes open to fail if the named file already exists.
O_TRUNC	if the file exists discardsthe file content and set the file size to zerobytes.
O_NONBLOCK	specifies that any subsequent read or write on the file should be nonblocking
O_NOCTTY	specifies not to use the named terminal device file as the calling process control terminal.

The umask function takes a new umask value as an argument.this new umask vale will be Used by the calling process from then on,and the function returns the old umask value. For example the following statement assigns the current umask value to the variable old_mask and sets the new umask value to " no execute for group" and " no write execute for others"

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

2. Explain how fcntl API is used for file and record locking. (08 marks)

(May-06)

Ans:

int fcntl(int fdesc, int cmd_flag, …);

The fdesc argument is a file descriptor for a file to be processed .The cmd_flag argument defines which operation is to be performed.

Cmd_flag	**use**
F_SETLK	sets a file lock. Do not block if this can't succeed immediately
F_SETLKW	sets a file lock and blocks th calling process until the lock is acquired.
F_GETLK	queries as to which process locked a specified region of a file.

For the file locking 3 rd argument is an address of a struct flock-typed variable .this specifies a region of a file where the lock is to be set,unset,or queried.

```
Struct flock{
Short   l_type; /*what lock to be set or to unlock file*\
Short   l_whence; /* a reference address for the next field*\
Off_t   l_start;/*offset fron 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*\
};
```

l_type value	**use**
F_RDLCK	sets a read lock on a specified region
F_WRLCK	sets a write lock on a specified region
F_UNLCK	unlocks a specified region

The l_whence ,l_start and l_len define a region of file to be locked/unlocked. L_whence defines a reference address to which l_start byte offset value is added.

l_whence value	**use**
SEEK_CUR	the l_start value is added to the current file pointer address
SEEK_SET	the l_start value is added to the byte zero of the file
SEEK_END	the l_start value is added to the end size of the file

if l_len is 0 ,the locked region extends from it's start address to a system impose limit on the maximum size of any file .This means that as the file size increases the lock also applies to the extended file region.
A struct flock type is defined and set by a process before it is passed to a fcntl call.

3. List the structure used to query the file attributes in Unix. Write a program in c++ to list the following file attributes of a given regular file passed as command line argument i) file type ii)hard link count iii)file size iv)file name. (08 marks)

Ans: **(May -06)**

```
Struct stat {
Dev_ts      t_dev;
Ino_t       st_ino;
Mode_t      st_mode;
Nlink_t     st_nlink;
Uid_t       st_uid;
Gid_t       st_gid;
Dev_t       st_rdev;
Off_t       st_size;
Time_t      st_atime;
Time_t      st_mtime;
Time_t      st_ctime;
};
```

```
#include<iostream.h>
#include<unistd.h>
#include<sys/stat.h>
Static char xtbl[10]=”rwxrwxrwx”;
#ifndef   MAJOR
#define   MINOR_BITS      8
#define   MAJOR(dev)       ((unsigned)dev>>MINOR_BITS)
#define   MINOR(dev)          (dev & MINOR_BITS)\

Static void display_file_type(int st_mode)
{
Switch(st_mode & S_IFMT)
{
Case S_IFDIR:  cout<<’d’;return;
Case S_IFCHR:  cout<<’c’;return;
Case S_IFBLK:  cout<<’b’;return;
Case S_IFREG:  cout<<’-’;return;
Case S_IFLNK:  cout<<’l’;return;
Case S_IFFIFO:  cout<<’p’;return;
}
}

static void display_access_perm(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] : ‘-‘;
Cout<<amode;
}
```

```
Static void long_list(char *path_name)
{
Struct stat statv;
Struct group *gr_p;
Struct passwd  *pw_p;
If (lstat(path_name,&statv))
{
Cout<<"invalid "<<endl;
Return;
}
Display_file_type(st_mode);
Display_access_perm(st_mode);
Cout<<statv.st_nlink;
Gr_p=getgrgid(statv.st_gid);
Pw_p=getpwuid(statv.st_uid);
Cout<<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)
Cout<<MAJOR(statv.st_rdev)<<MINOR(statv.st_rdev);
Else
Cout<<statv.st_size;
Cout<<ctime(&statv.st_mtime);
Cout<<path_name<<endl;
}
Int main(int argc,char * argv[])
{
While(--argc >= 1) long_list(*++argv);
```

Return 0;

}

4. Describe the open API, clearly its prototype declaration indicating its prototype declaration, the values the arguments take along with their meaning. Give two instances when open API can fail. List all the access modifier flags and explain their meanings. (06 marks) (Dec-06/May-08)

Ans:

Int open(const char * path_name,int access_mode,mode_t permission);

Path_name is the path name of the file,it can be absolute or relative name.

Access mode flag	**use**
O_RDONLY	opens the file for read only
O_WRONLY	opens the file for write only
O_RDWR	opens the file for read and write
O_APPEND	appends data to the end of the file
O_CREAT	creates the file if it does't exists

O_EXCL used with the O_CREAT only. This flag causes open to fail if the named file already exists.

O_TRUNC if the file exists discardsthe file content and set the file size to zerobytes.

O_NONBLOCK specifies that any subsequent read or write on the file should be nonblocking

O_NOCTTY specifies not to use the named terminal device file as the calling process control terminal.

If the named file does't exists and the O_CREAT flag is not specified then open will abort with a failure return status.when both O_CREAT & O_EXCL flags are specified the open will fail if the named file exists.

5. List the important uses of fcnt API. Give its prototype description. Write a c++ program to check whether the close – on – exec flag is set for a given file. If it is not set, use fcntl to set this flag. Also show the implementation of dups macro using this API. (06 marks) (May-08)

Ans:

Used to query or set access control flags and the close on exec flag of any file descriptor.used to assign multiple file descriptor to reference the same file. Used in file and record locking.

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

{

int  fd=Open(argv[1],O_RDONLY);

If(fcntl(fd,F_GETTFD) == 0)

(Void)fcntl(fdesc,F_SETFD,1);

return 0;

}
```

Dup2 implementation

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

6.Write a program to implement ls –l command (10 marks) (May-10)

Ans:

```
#include<iostream.h>

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

```
Static char xtbl[10]=”rwxrwxrwx”;
#ifndef   MAJOR
#define   MINOR_BITS      8
#define   MAJOR(dev)       ((unsigned)dev>>MINOR_BITS)
#define   MINOR(dev)          (dev & MINOR_BITS)\

Static void display_file_type(int st_mode)
{
Switch(st_mode & S_IFMT)
{
Case S_IFDIR:  cout<<’d’;return;
Case S_IFCHR:  cout<<’c’;return;
Case S_IFBLK:  cout<<’b’;return;
Case S_IFREG:  cout<<’-’;return;
Case S_IFLNK:  cout<<’l’;return;
Case S_IFFIFO:  cout<<’p’;return;
}
}

static void display_access_perm(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] : ‘-‘;
Cout<<amode;
}

Static void long_list(char *path_name)
```

```
{
Struct stat statv;
Struct group *gr_p;
Struct passwd  *pw_p;
If (lstat(path_name,&statv))
{
Cout<<"invalid "<<endl;
Return;
}
Display_file_type(st_mode);
Display_access_perm(st_mode);
Cout<<statv.st_nlink;
Gr_p=getgrgid(statv.st_gid);
Pw_p=getpwuid(statv.st_uid);
Cout<<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)
Cout<<MAJOR(statv.st_rdev)<<MINOR(statv.st_rdev);
Else
Cout<<statv.st_size;
Cout<<ctime(&statv.st_mtime);
Cout<<path_name<<endl;
}
Int main(int argc,char * argv[])
{
While(--argc >= 1) long_list(*++argv);
Return 0;
}
```

UNIT - 4

UNIX PROCESSES

1. **Write an explanatory note on environment variables. Also write a C/C++ program that outputs the contents of its environment list. (06 marks)**

Ans: **(Dec-06)**

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

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

We'll call *environ* the environment pointer, the array of pointers the environment list, and the strings they point to the environment strings. By convention, the environment consists of name=value strings,

C/C++ program that outputs the contents of its environment list.

```
#include "apue.h"
int
main(int argc, char *argv[])
{
   int        i;
   char       **ptr;
   extern char **environ;

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

   exit(0);
}
```

2. With an example explain the use of setjmp and longjmp functions. (08 marks)

Ans: (May-07)

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

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

Returns: 0 if called directly, nonzero if returning from a call to longjmp
void longjmp(jmp_buf env, int val);

We call setjmp from the location that we want to return to, which in this example is in the main function. In this case, setjmp returns 0 because we called it directly. In the call to setjmp, the argument env is of the special type jmp_buf. This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp. Normally, the env variable is a global variable, since we'll need to reference it from another function.

When we encounter an error say, in the cmd_add functionwe call longjmp with two arguments. The first is the same env that we used in a call to setjmp, and the second, val, is a nonzero value that becomes the return value from setjmp. The reason for the second argument is to allow us to have more than one longjmp for each setjmp. For example, we could longjmp from cmd_add with a val of 1 and also call longjmp from get_token with a val of 2. In the main function, the return value from setjmp is either 1 or 2, and we can test this value, if we want, and determine whether the longjmp was from cmd_add or get_token.

Let's return to the example. Example shows both the main and cmd_add functions. (The other two functions, do_line and get_token, haven't changed.)

Example of setjmp and longjmp

```
#include "apue.h"
#include <setjmp.h>
#define TOK_ADD    5
jmp_buf jmpbuffer;
int
main(void)
{
     char    line[MAXLINE];
     if (setjmp(jmpbuffer) != 0)
         printf("error");
     while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
     exit(0);
}
 ...
void
cmd_add(void)
{
    int     token;
    token = get_token();
    if (token < 0)    /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

When main is executed, we call setjmp, which records whatever information it needs to in the variable jmpbuffer and returns 0. We then call do_line, which calls cmd_add, and assume that an error of some form is detected. longjmp causes the stack to be "unwound" back to the main function, throwing away the stack frames for cmd_add and do_line

Calling longjmp causes the setjmp in main to return, but this time it returns with a value of 1 (the second argument for longjmp).

3. Describe the UNIX Kernel support for process. Show the related data structures. (06 marks) (May-12)

Ans:

Unix process consists of text segment, data segment,stack segmenta unix kernel has a process table that keeps track of all active processes.some of the processes belong to the kernel.they are called system processes.the majority of processesare associate with the users who are logged in.each entry in the process table contains pointer to the text,data,satck segment and the u-area of a process.the u-area is an extension of a process table entry and contains other process specific data,such as FDT,current root and working directory and inode numbers and set of system imposed process resource limits.

After a fork system for both parent and child process resumee execution at a return of fork function.when a process is created by fork to contains dubplicated copy of text,data,stack segments of its parent.also it has a FDT contains references to the same opened files as its parent.the process is assigned with the attributes whitch are inherited by its parent or set by kernel like

rUID: the user id of a user who created parent process.

rGID: the group id of a user who created parent process.

eUID: same as rUID expect when the file that was executed to create the process has its set UID flag turned on.

eGID: same as rGID expect when the file that was executed to create the process has its set GID flag turned on.

Saved set_UID and Saved set_GID: these are the assigned eUID and Egid respectively of the process.

PGID,SID,current directory,root directory,signal handling,signal mask,umask,nice value,controlling terminal.

The attrubitues whitch differ are PID,PPID ,pending signals,alarum clock time and file locks.

4. Bring out the importance of locking files. What is the drawback of advisory lock? Explain in brief. (05 marks) (Dec-08/May-09)

Ans:

UNIX systems allow multiple processes to read and write the same file concurrently. This provides a means for data sharing among process but it also renders difficulty for any prosess in determining when data in a file can be overridden by another process.This is especially important for application like a data base mamager. To remedy this draw back UNIX supports file locking.

File locks are mandatory if they are enforce by an OS kernel.If a mandatory exclusive lock is set on a file no process can use the read/write call to access data on the locked region. If a mandatory shared lock is set on a file no process can use the write call to access data on the locked region.

An advisory lock is not enforced by a kernel at the system call level this means even though a lock read/write may be set on afile other process can still use the read/write call to access the file.the procedure for lock is

Set the lock if failed,wait for lock request gained.then read/write and then release the lock.

As each process should follow this this lock is safe.but it has drawback is that to follow the procedure is difficult as the programs are obtained from different sources.

5. In a certain application,it is required to lock the hatched portion of the file as shown in fig.4(b). before locking the program must query the OS to see if some other process has locked the file. If yes, give the details of the locked portion and the PID of the process.once the lock is obtained perform a write and unlock the file.

Write a c++ program to implement this application. Assume suitable lock type. (06 marks) (May-05)

6. What are the different ways in which a process can terminate? With a neat block schematic, explain how a process is launched and terminates clearly indicating the role of C-startup routine and the exit handlers. (05 marks) (Dec-06/May-08)

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

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

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

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

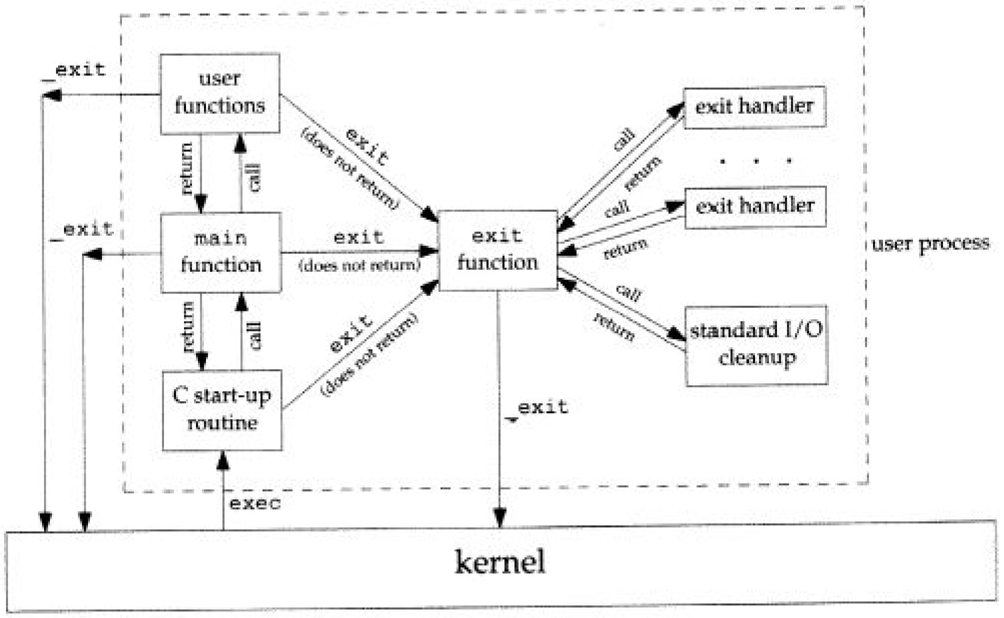
There are eight ways for a process to terminate. Normal termination occurs in five ways:

1. Return from main
2. Calling exit
3. Calling _exit or _Exit
4. Return of the last thread from its start routine
5. Calling pthread_exit from the last thread

Abnormal termination occurs in three ways:

6. Calling abort
7. Receipt of a signal
8. Response of the last thread to a cancellation request

A process can register up to 32 functions that are automatically called by exit. These are called exit handlers and are registered by calling the atexit function. We pass the address of a function as the argument to atexit. When this function is called, it is not passed any arguments and is not expected to return a value. The exit function calls these functions in reverse order of their registration. Each function is called as many times as it was registered. exit first calls the exit handlers and then closes (via fclose) all open streams.

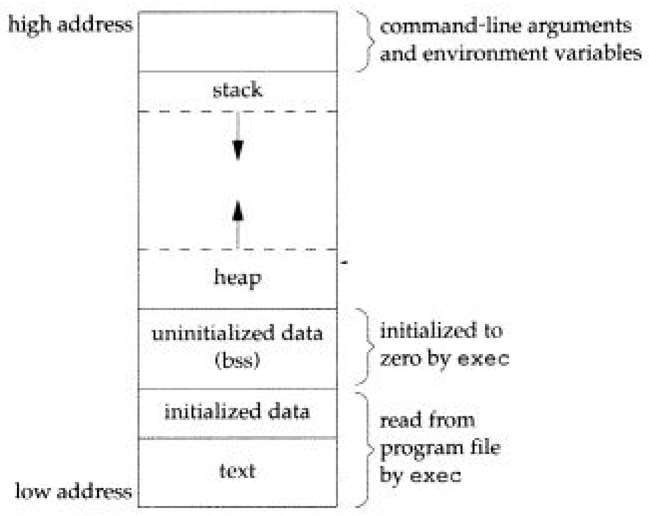


7. With a neat diagram, explain the memory layout of c program. In which segments are the automatic variables and dynamically created objects are stored?
(04 marks) (May-10/Dec-11)

Ans: C program has been composed of the following pieces:

- Text segment, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

- Initialized data segment, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration int maxcount = 99;
  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
- Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration long sum[1000];
  appearing outside any function causes this variable to be stored in the uninitialized data segment.
- Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables.
- Heap, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.


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

The memory layout of a c program

9. **What are setjmp and longjmp function? Explain with a program to transfer the control across functions using them. (10 marks) (May-12)**

Ans:

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

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

Returns: 0 if called directly, nonzero if returning from a call to longjmp
void longjmp(jmp_buf env, int val);

We call setjmp from the location that we want to return to, which in this example is in the main function. In this case, setjmp returns 0 because we called it directly. In the call to setjmp, the argument env is of the special type jmp_buf. This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp. Normally, the env variable is a global variable, since we'll need to reference it from another function.

When we encounter an error say, in the cmd_add functionwe call longjmp with two arguments. The first is the same env that we used in a call to setjmp, and the second, val, is a nonzero value that becomes the return value from setjmp. The reason for the second argument is to allow us to have more than one longjmp for each setjmp. For example, we could longjmp from cmd_add with a val of 1 and also call longjmp from get_token with a val of 2. In the main function, the return value from setjmp is either 1 or 2, and we can test this value, if we want, and determine whether the longjmp was from cmd_add or get_token.

Let's return to the example. Example shows both the main and cmd_add functions. (The other two functions, do_line and get_token, haven't changed.)

Example of setjmp and longjmp

```
#include "apue.h"
#include <setjmp.h>
#define TOK_ADD    5
jmp_buf jmpbuffer;
int
main(void)
{
    char    line[MAXLINE];
   if (setjmp(jmpbuffer) != 0)
       printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
      do_line(line);
    exit(0);
}
```

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

When main is executed, we call setjmp, which records whatever information it needs to in the variable jmpbuffer and returns 0. We then call do_line, which calls cmd_add, and assume that an error of some form is detected. longjmp causes the stack to be "unwound" back to the main function, throwing away the stack frames for cmd_add and do_line Calling longjmp causes the setjmp in main to return, but this time it returns with a value of 1 (the second argument for longjmp).

UNIT - 5

PROCESS CONTROL

1. Explain the following system calls:

i)fork ii)vfork iii)exit iv)wait (10 marks) (May-08)

Ans:

pid_t fork(void);
Returns: 0 in child, process ID of child in parent, 1 on error

The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

Both the child and the parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory. The parent and the child share the text segment

pid_t vfork(void);
Returns: 0 in child, process ID of child in parent, 1 on error

The vfork function is intended to create a new process when the purpose of the new process is to exec a new program (step 2 at the end of the previous section). The bare-bones shell in the program from Figure 1.7 is also an example of this type of program. The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls exec (or exit) right after the vfork. Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent.

Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes. (This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)

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

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

We'll discuss the effect of these three functions on other processes, such as the children and the parent of the terminating process.

The reason for the different headers is that exit and _Exit are specified by ISO C, whereas _exit is specified by POSIX.1.

Historically, the exit function has always performed a clean shutdown of the standard I/O library: the fclose function is called for all open streams. This causes all buffered output data to be flushed (written to the file).

All three exit functions expect a single integer argument, which we call the exit status. Most UNIX System shells provide a way to examine the exit status of a process. If (a) any of these functions is called without an exit status, (b) main does a return without a return value, or (c) the main function is not declared to return an integer, the exit status of the process is undefined. However, if the return type of main is an integer and main "falls off the end" (an implicit return), the exit status of the process is 0.

The exit status was undefined if the end of the main function was reached without an explicit return statement or call to the exit function.

Wait:

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent.

we need to be aware that a process that calls wait or waitpid can

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

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

pid_t wait(int *statloc);
Both return: process ID if OK, 0 (see later), or 1 on error

The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.

The argument statloc is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument. If we don't care about the termination status, we simply pass a null pointer as this argument.

2. Giving the prototype explain different variant of exec system call. (10 marks)

Ans: (Dec-05/May-09)

Use of the fork function is to create a new process (the child) that then causes another program to be executed by calling one of the exec functions. When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current processits text, data, heap, and stack segmentswith a brand new program from disk.

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

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

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

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

The PATH variable contains a list of directories, called path prefixes, that are separated by colons.

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

The final command-line argument is followed by a null pointer. If this null pointer is specified by the constant 0, we must explicitly cast it to a pointer; if we don't, it's interpreted as an integer argument. If the size of an integer is different from the size of a char *, the actual arguments to the exec function will be wrong.

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

3. What is race condition? Write a program in C/C++ to illustrate a race condition. (06 marks) (Dec-08)

Ans: For our purposes, a race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run. The fork function is a lively breeding ground for race conditions, if any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork. In general, we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

Program with a race condition

```
#include "apue.h"
static void charatatime(char *);
int
main(void)
{
   pid_t   pid;
   if ((pid = fork()) < 0) {
      err_sys("fork error");
   } else if (pid == 0) {
      charatatime("output from child\n");
   } else {
      charatatime("output from parent\n");
   }
   exit(0);
}
static void
charatatime(char *str)
{
   char    *ptr;
   int     c;
   setbuf(stdout, NULL);           /* set unbuffered */
   for (ptr = str; (c = *ptr++) != 0; )
      putc(c, stdout);
}
```

Output:

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

4. How UNIX operating system keeps process accounting? (06 marks) (Dec-07)

Ans:

Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates. These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on.

A function we haven't described (acct) enables and disables process accounting. The only use of this function is from the accton(8) command (which happens to be one of the few similarities among platforms). A superuser executes accton with a pathname argument to enable accounting. The accounting records are written to the specified file, which is usually /var/account/acct

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

```
typedef  u_short comp_t;   /* 3-bit base 8 exponent; 13-bit fraction */
struct  acct
{
  char   ac_flag;     /* flag (see Figure 8.26) */
  char   ac_stat;     /* termination status (signal & core flag only) */
```

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

The ac_flag member records certain events during the execution of the process. These events are described in Figure.

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

ac_flag	Description
AEXPND	expanded accounting entry

The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork. Each accounting record is written when the process terminates. This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started

To perform our test, we do the following:

1. Become superuser and enable accounting, with the accton command. Note that when this command terminates, accounting should be on; therefore, the first record in the accounting file should be from this command.
2. Exit the superuser shell and run the program to test. This should append six records to the accounting file: one for the superuser shell, one for the test parent, and one for each of the four test children. A new process is not created by the execl in the second child. There is only a single accounting record for the second child.
3. Become superuser and turn accounting off. Since accounting is off when this accton command terminates, it should not appear in the accounting file.
4. Run another program to print the selected fields from the accounting file.

5. What is job control? Summarize the job control features with the help of a figure. (08 marks)

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

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

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

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

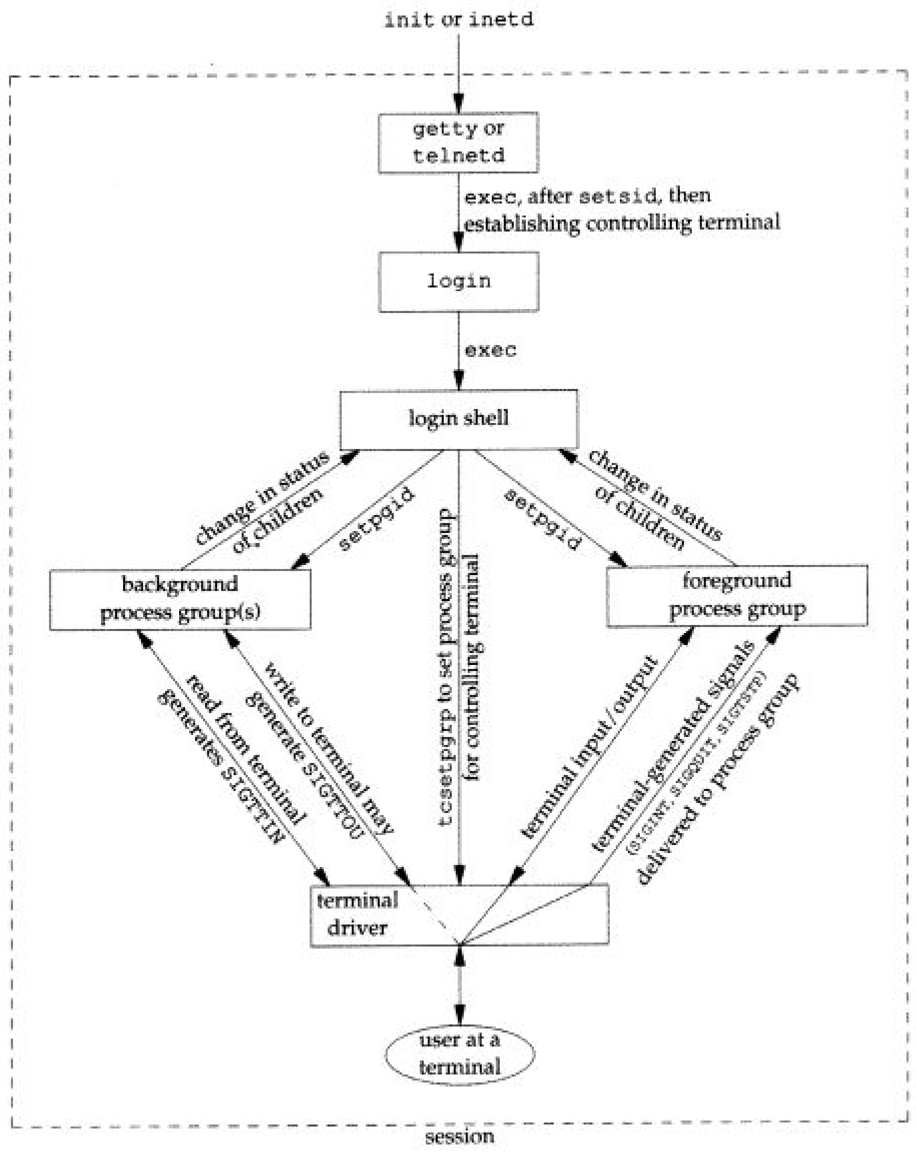
Another job control condition can arise that must be handled by the terminal driver. Since we can have a foreground job and one or more background jobs, which of these receives the characters that we enter at the terminal? Only the foreground job receives terminal input. It is not an error for a background job to try to read from the terminal, but the terminal driver detects this and sends a special signal to the background job: SIGTTIN. This signal normally stops the background job; by using the shell, we are notified of this and can bring the job into the foreground so that it can read from the terminal. The following demonstrates this:

```
$ cat > temp.foo &         start in background, but it'll read from standard input
[1]   1681
$                          we press RETURN
[1] + Stopped (SIGTTIN)    cat > temp.foo &
$ fg %1                    bring job number 1 into the foreground
cat > temp.foo             the shell tells us which job is now in the foreground
  hello, world             enter one line
  ^D                    type the end-of-file character
$ cat temp.foo           check that the one line was put into the file
hello, world
```

The shell starts the cat process in the background, but when cat tries to read its standard input (the controlling terminal), the terminal driver, knowing that it is a background job, sends the SIGTTIN signal to the background job. The shell detects this change in status of its child and tells us that the job has been stopped. We then move the stopped job into the foreground with the shell's fg command. Doing this causes the shell to place the job into the foreground process group (tcsetpgrp) and send the continue signal (SIGCONT) to the process group. Since it is now in the foreground process group, the job can read from the controlling terminal.

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

```
$ cat temp.foo &           execute in background
[1]    1719
$ hello, world             the output from the background job appears after the prompt
                        we press RETURN
[1] + Done              cat temp.foo &
  $ stty tostop              disable ability of background jobs to output to
 controlling terminal
$ cat temp.foo &           try it again in the background
[1]    1721
```


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

```
$ we press RETURN and find the job is stopped
  [1] + Stopped(SIGTTOU)          cat temp.foo &
  $ fg %1                  resume stopped job in the foreground
  cat temp.foo              the shell tells us which job is now in the foreground
  hello, world              and here is its output
```

When we disallow background jobs from writing to the controlling terminal, cat will block when it tries to write to its standard output, because the terminal driver identifies the write as coming from a background process and sends the job the SIGTTOU signal. As with the previous example, when we use the shell's fg command to bring the job into the foreground, the job completes.

The Figure summarizes some of the features of job control that we've been describing. The solid lines through the terminal driver box mean that the terminal I/O and the terminal-generated signals are always connected from the foreground process group to the actual terminal. The dashed line corresponding to the SIGTTOU signal means that whether the output from a process in the background process group appears on the terminal is an option.

7. With a prototype description of fork, explain the special features of this API. Write a program to create a child process and print the PPID and PID in the child process. The parent process must ensure that the child does not become a Zombie process. The parent process must wait for the child and print exit status of the child using appropriate macros. (06 marks) (Dec-09)

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

pid_t fork(void);
Returns: 0 in child, process ID of child in parent, 1 on error

The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

```
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));
}
int
main(void)
{
    pid_t   pid;
    int status;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {     /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */
        /*
         * We're the second child; our parent becomes init as soon
         * as our real parent calls exit() in the statement above.
         * Here's where we'd continue executing, knowing that when
         * we're done, init will reap our status.
         */
        sleep(2);
        printf("second child pid = %d, parent pid = %d\n", pid, getppid());
        exit(0);
    }
    if (waitpid(pid, &ststus, 0) != pid)  /* wait for first child */
        err_sys("waitpid error");
    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
pr_exit(status);
    exit(0);
}
```

8. Explain in brief, what happens when exec is called in a child process. List the 6 different forms of exec APIs. Write a program that execs a program echo all to display all the command line and environment variables when this program is exceed in the child process space. (06 marks) (May-09/Dec-10)

Ans:

one use of the fork function is to create a new process (the child) that then causes another program to be executed by calling one of the exec functions. When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process its text, data, heap, and stack segments with a brand new program from disk.

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


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

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

```
char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int
main(void)
{
   pid_t   pid;
   if ((pid = fork()) < 0) {
       err_sys("fork error");
   } else if (pid == 0) {  /* specify pathname, specify environment */
       if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
               "MY ARG2", (char *)0, env_init) < 0)
           err_sys("execle error");
   }
   if (waitpid(pid, NULL, 0) < 0)
       err_sys("wait error");
```

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

The echo all program:

```
int
main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;
    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);
    exit(0);
}
```

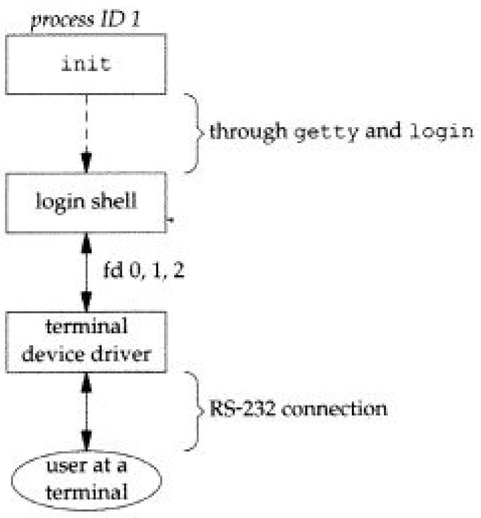
9. With a neat block schematic, explain the terminal login process in BSD Unix. What is a session? Explain how do you create a session using appropriate shell commands. (04 marks) (May-08/Dec-09)

Ans:

When the system is bootstrapped, the kernel creates process ID 1, the init process, and it is init that brings the system up multiuser. The init process reads the file /etc/ttys and, for every terminal device that allows a login, does a fork followed by an exec of the program getty. It is getty that calls open for the terminal device. The terminal is opened for reading and writing. If the device is a modem, the open may delay inside the device driver until the modem is dialed and the call is answered. Once the device is open, file descriptors 0, 1, and 2 are set to the device. Then getty outputs something like login: and waits for us to enter our user name.

When we enter our user name, getty's job is complete, and it then invokes the login program The login program does many things. Since it has our user name, it can call getpwnam to fetch our password file entry. Then login calls getpass(3) to display the prompt Password: and read our password (with echoing disabled, of course). It calls crypt(3) to encrypt the password that we entered and compares the encrypted result to the pw_passwd field from our shadow password file entry. If the login attempt fails because of an invalid password (after a few tries), login calls exit with an argument of 1. This termination will be noticed by the parent (init), and it will do another fork followed by an exec of getty, starting the procedure over again for this terminal.

At this point, our login shell is running. Its parent process ID is the original init process (process ID 1), so when our login shell terminates, init is notified (it is sent a SIGCHLD signal), and it can start the whole procedure over again for this terminal. File descriptors 0, 1, and 2 for our login shell are set to the terminal device. Figure shows this arrangement.



A session is a collection of one or more process groups.

For example, the session could have been generated by shell commands of the form

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

10. What is job control? What are the three forms of support from the OS required for job control? (04 Marks) (Dec-11)

Ans:

Job control is a feature added to BSD around 1980. This feature allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are to run in the background. Job control requires three forms of support:

1. A shell that supports job control
2. The terminal driver in the kernel must support job control
3. The kernel must support certain job-control signal

UNIT - 6

SIGNALS AND DAEMON PROCESSES

1. **What is a signal? Discuss any five POSIX defined signals. Explain how setup signal handler. (10 marks) (May-07/Dec-09)**

Ans:

Signals are triggred by events and are posted on a process to notify it that something has happened and requires some action.An event can be generated from a process, a user or the kernel.

Signal name	**use**	**core file generated at default**
SIGALRM	alarm timer time outs.can be generated by alarm()API.	No
SIGABRT	abort process execution can be generated by abort () API.	Yes
SIGFPE	illegal mathematical operation.	Yes
SIGHUP	controlling terminal hangup.	No
SIGILL	execution of an illegal mechine instruction.	Yes

The signal API can be used to define the per signal handling method.

Void (*signal (int signal_num, void (* handler)(int)))(int);

The formal argument of an API is signal_num and is signal identifier like SIGINT or SIGTERM. The handler argument is functiuon pointer of a user define signal handler function. This function should take an integer formal argument and does not return anything.

The example catches SIGTERM signals and ignores SIGINT signals and accept default action of SIGSEGE signal.the pause API susoends the calling process until it is interrupted by a signal and the corresponding signal handler does return.

```
#include<iostream.>
#include<signal.h>
/* signal handler function*/
```

```
Void catch_sig( int sig_num )
{
  Signal(sig_num, catch_sig )
Cout<<"catch_sig:"<<sig_num<<endl;
}
/*main function*/
Int main()
{
Signal(SIGTERM,catch_sig);
Signal(SIGINT,SIG_IGN);
Ignal(SIGSEGV,SEG_DFL);
Pause();
}
```

2. What is a daemon? Discuss the basic coding rules. (10 marks) (May-06/Dec-08)

Ans: 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. Because they don't have a controlling terminal, we say that they run in the background.

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

1. The first thing to do is call umask to set the file mode creation mask to 0. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions. For example, if it specifically creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts.
2. Call fork and have the parent exit. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the

shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to setsid that is done next.

3. Call setsid to create a new session. The three steps listed in Section 9.5 occur. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.
4. 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. Alternatively, some daemons might change the current working directory to some specific location, where they will do all their work. For example, line printer spooling daemons often change to their spool directory.
5. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent (which could be a shell or some other process). We can use our open_max function or the getrlimit functionm to determine the highest descriptor and close all descriptors up to that value.
6. 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.

3. **What are signals? Mention the different sources of signals. What are the three dispositions the process has when signals occur? List any four signals along with one or two line explanation. Wrtie a program to setup handlers for SIGINT and SIGACARM signals. (08 marks) (May-10/Dec-11)**

Ans:

Signals are triggred by events and are posted on a process to notify it that something has happened and requires some action.An event can be generated from a process, a user or the kernel.

Signal name	use	core file generated at default
SIGALRM	alarm timer time outs.can be generated by alarm()API.	No
SIGABRT	abort process execution can be generated by abort () API.	Yes
SIGFPE	illegal mathematical operation.	Yes
SIGHUP	controlling terminal hangup.	No
SIGILL	execution of an illegal mechine instruction.	Yes

4. **What are daemon processes? Enlist their characteristics. Also write a program to transform a normal user process into a daemon process. Explain every step in the program. (06 marks) (May-08)**

Ans: 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. Because they don't have a controlling terminal, we say that they run in the background.

```
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
```

```
#include "apue.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)
   err_quit("%s: can't get file limit", cmd);
/*
 * Become a session leader to lose controlling TTY.
 */
if ((pid = fork()) < 0)
   err_quit("%s: can't fork", cmd);
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)
   err_quit("%s: can't ignore SIGHUP");
if ((pid = fork()) < 0)
   err_quit("%s: can't fork", cmd);
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)
            err_quit("%s: can't change directory to /");
        /*
         * 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);
  }
#include "apue.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)
        err_quit("%s: can't get file limit", cmd);
    /*
     * Become a session leader to lose controlling TTY.
     */
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    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)
    err_quit("%s: can't ignore SIGHUP");
if ((pid = fork()) < 0)
    err_quit("%s: can't fork", cmd);
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)
    err_quit("%s: can't change directory to /");
/*
 * 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);
    }
}
```

UNIT - 7

INTERPROCESS COMMUNICATION

1. **What are pipes? What are their limitations? Write a c program that sends "hello world" message to the child process through the pipe. The child on receiving this message should display it on the standard output.**
   (06 marks) **(Dec-05/08)**

Ans:

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

1. 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.
2. 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.

```
#include "apue.h"
int
main(void)
{
   int     n;
   int     fd[2];
   pid_t   pid;
   char    line[MAXLINE];
   if (pipe(fd) < 0)
      err_sys("pipe error");
   if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

2. **With a neat block schematic, explain how FIFO can be used to implement client-server communication model. (04 marks) (May-07)**

Ans: Client-Server Communication Using a FIFO

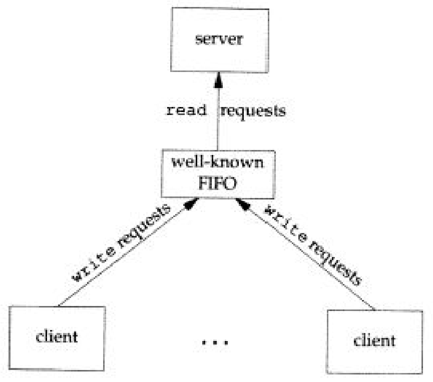
If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. "well-known" means that the pathname of the FIFO is known to all the clients that need to contact the server. The Figure shows this arrangement. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size. This prevents any interleaving of the client writes.

The problem in using FIFOs for this type of client-server communication is how to send replies back from the server to each client. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client's process ID. For example, the server can create a FIFO with the name /tmp/serv1.XXXXX, where XXXXX is replaced with the client's process ID.

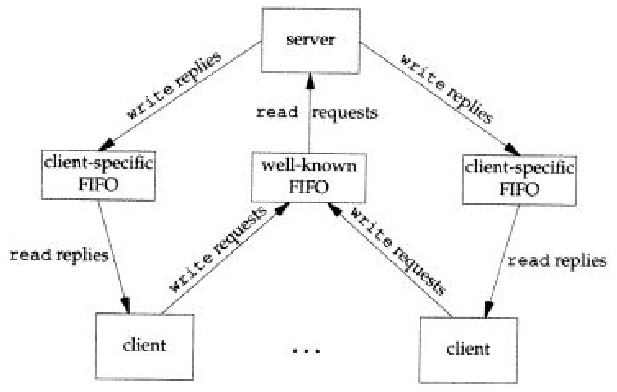
This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the file system. The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

With the arrangement shown in figure, 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.

Clients sending requests to a server using a FIFO



Clientserver communication using FIFOs



3. What are the three different ways in which the client and server processes are can get access to same IPC structures? List the APIs with their argument details that are used to create, control, send and receive messages from a message queue. (07 marks) (Dec-08)

Ans:

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

1. The server can create a new IPC structure by specifying a key of IPC_PRIVATE and store the returned identifier somewhere (such as a file) for the client to obtain. The key IPC_PRIVATE guarantees that the server creates a new IPC structure. The disadvantage to this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.

   The IPC_PRIVATE key is also used in a parentchild relationship. The parent creates a new IPC structure specifying IPC_PRIVATE, and the resulting identifier is then available to the child after the fork. The child can pass the identifier to a new program as an argument to one of the exec functions.

2. 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.
3. 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.

The function msgget to either open an existing queue or create a new queue.

int msgget(key_t key, int flag);
Returns: message queue ID if OK, 1 on error

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

The flag argument an integer flag. The options for flag can IPC_PRIVATE ,IPC_CREAT, IPC_EXCL, IPC_NOWAIT

The msgctl function performs various operations on a queue.

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
Returns: 0 if OK, 1 on error

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

IPC_STAT	Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf.
IPC_SET	
IPC_RMID	Remove the message queue from the system and any data still on the queue. This removal is immediate

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

IPC_STAT	Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf.

Returns: 0 if OK, 1 on error

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

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

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

```
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

As with msgsnd, the ptr argument points to a long integer followed by a data buffer for the actual message data. nbytes specifies the size of the data buffer. 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

type==0 The first message on the queue is returned.

than or equal to the absolute value of type is returned.

4. What are semaphores? What is their purpose? List and explain the APIs used to create and control the semaphores. (03 marks) (May-07/10)

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

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

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 */);

Returns: (see following)

5. What are the different system calls available to create and manipulate semaphores? Explain. (10 marks) (Dec-08/May-10/Dec-11)

Ans:

The function to create 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

Whenever an IPC structure is being created (by calling msgget, semget, or shmget), a key must be specified. The data type of this key is the primitive system data type key_t,. This key is converted into an identifier by the kernel.

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

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

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

The semctl function is the catchall for various semaphore operations.

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int  cmd,
          ... /* union semun arg */);
```

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

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

Note that the optional argument is the actual union, not a pointer to the union.

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

IPC_STAT	Fetch the semid_ds structure for this set, storing it in the structure pointed to by arg.buf.
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.

For all the GET commands other than GETALL, the function returns the corresponding value. For the remaining commands, the return value is 0.

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

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[],
    size_t nops);
                              Returns: 0 if OK, 1 on error
```

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

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

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

The operation on each member of the set is specified by the corresponding sem_op value. This value can be negative, 0, or positive. (In the following discussion, we refer to the "undo" flag for a semaphore. This flag corresponds to the SEM_UNDO bit in the corresponding sem_flg member.)

1. 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.
2. 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.

3. 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.

The semop function operates atomically; it does either all the operations in the array or none of them.

UNIT - 8

NETWORK IPC: SOCKETS

1. What is a socket? Discuss how it create and destroy a socket? (May-10)

Ans:

A socket is an abstraction of a communication endpoint. Just as they would use file descriptors to access a file, applications use socket descriptors to access sockets. Socket descriptors are implemented as file descriptors in the UNIX System. Indeed, many of the functions that deal with file descriptors, such as read and write, will work with a socket descriptor.

To create a socket, we call the socket function.

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Returns: file (socket) descriptor if OK, 1 on error

The domain argument determines the nature of the communication, including the address format The constants start with AF_ (for address family) because each domain has its own format for representing an address.

Domain	Description
AF_INET	IPv4 Internet domain
AF_INET6	IPv6 Internet domain
AF_UNIX	UNIX domain
AF_UNSPEC	unspecified

The type argument determines the type of the socket, which further determines the communication characteristics.

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

The protocol argument is usually zero, to select the default protocol for the given domain and socket type. When multiple protocols are supported for the same domain and socket type, we can use the protocol argument to select a particular protocol. The default protocol for a SOCK_STREAM socket in the AF_INET communication domain is TCP (Transmission Control Protocol). The default protocol for a SOCK_DGRAM socket in the AF_INET communication domain is UDP (User Datagram Protocol).

We can disable I/O on a socket with the shutdown function.

```
#include <sys/socket.h>
int shutdown (int sockfd, int how);

                    Returns: 0 if OK, 1 on error
```

If how is SHUT_RD, then reading from the socket is disabled. If how is SHUT_WR, then we can't use the socket for transmitting data. We can use SHUT_RDWR to disable both data transmission and reception.

2. Discuss the different functions available for transmitting and receiving data over a socket. (10 marks) (May-10/Dec-11)

Ans:

The function for transmitting are

ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
Returns: number of bytes sent if OK, 1 on error

Sockfd is the socket descriptor, buf is a generic pointer referes to the location from where data had to be read to transfer, nbytes indicates the the number of bytes to be trnfered.

Flag	Description
MSG_DONTROUTE	Don't route packet outside of local network.
MSG_DONTWAIT	Enable nonblocking operation (equivalent to using O_NONBLOCK).
MSG_EOR	This is the end of record if supported by protocol.
MSG_OOB	Send out-of-band data if supported by protocol.

The sendto function is similar to send. The difference is that sendto allows us to specify a destination address to be used with connectionless sockets.

```
ssize_t sendto(int sockfd, const void *buf, size_t
    nbytes, int flags,
             const struct sockaddr *destaddr,
    socklen_t destlen);
```

Returns: number of bytes sent if OK, 1 on error

Dept.of CS&E,SJBIT

With a connection-oriented socket, the destination address is ignored, as the destination is implied by the connection. With a connectionless socket, we can't use send unless the destination address is first set by calling connect, so sendto gives us an alternate way to send a message.

We can call sendmsg with a msghdr structure to specify multiple buffers from which to transmit data

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
Returns: number of bytes sent if OK, 1 on error

POSIX.1 defines the msghdr structure to have at least the following members:

```
struct msghdr {
  void         *msg_name;        /* optional address */
  socklen_t     msg_namelen;      /* address size in bytes */
  struct iovec  *msg_iov;         /* array of I/O buffers */
  int           msg_iovlen;       /* number of elements in array */
  void         *msg_control;      /* ancillary data */
  socklen_t     msg_controllen;   /* number of ancillary bytes */
  int           msg_flags;        /* flags for received message */
  .
  .
  .
};
```

The recv function is similar to read, but allows us to specify some options to control how we receive the data.

ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
Returns: length of message in bytes, 0 if no messages are available and peer has done an orderly shutdown, or 1 on error

Flag	Description
MSG_OOB	Retrieve out-of-band data if supported by protocol
MSG_PEEK	Return packet contents without consuming packet.
MSG_TRUNC	Request that the real length of the packet be returned, even if it was truncated.
MSG_WAITALL	Wait until all data is available (SOCK_STREAM only).

If we are interested in the identity of the sender, we can use recvfrom to obtain the source address from which the data was sent.

ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags, struct sockaddr *restrict addr, socklen_t *restrict addrlen);
Returns: length of message in bytes, 0 if no messages are available and peer has done an orderly shutdown, or 1 on error

If addr is non-null, it will contain the address of the socket endpoint from which the data was sent. When calling recvfrom, we need to set the addrlen parameter to point to an integer containing the size in bytes of the socket buffer to which addr points. On return, the integer is set to the actual size of the address in bytes.

Because it allows us to retrieve the address of the sender, recvfrom is usually used with connectionless sockets. Otherwise, recvfrom behaves identically to recv.

To receive data into multiple buffers we can use recvmsg.

```
ssize_t recvmsg(int sockfd, struct msghdr *msg,
    int flags);
```

Returns: length of message in bytes, 0 if no messages are available and peer has done an orderly shutdown, or 1 on error

The msghdr structure (which we saw used with sendmsg) is used by recvmsg to specify the input buffers to be used to receive the data. We can set the flags argument to change the default behavior of recvmsg. On return, the msg_flags field of the msghdr structure is set to indicate various characteristics of the data received. (The msg_flags field is ignored on entry to recvmsg).

Flag	Description
MSG_CTRUNC	Control data was truncated.
MSG_DONTWAIT	recvmsg was called in nonblocking mode.
MSG_EOR	End of record was received.
MSG_OOB	Out-of-band data was received.
MSG_TRUNC	Normal data was truncated.

3. **What is out-of-band data? Illustrate with example, how would you specify out-of-band data. (03 marks) (Dec-07)**

Ans:

Out-of-band data is an optional feature supported by some communication protocols, allowing higher-priority delivery of data than normal. Out-of-band data is sent ahead of any data that is already queued for transmission. TCP refers to out-of-band data as "urgent" data. TCP supports only a single byte of urgent data, but allows urgent data to be delivered out of band from the normal data delivery mechanisms. To generate urgent data, we specify the MSG_OOB flag to any of the three send functions. If we send more than one byte with the MSG_OOB flag, the last byte will be treated as the urgent-data byte.

4.Write short notes on :

i) race condition

ii) error logging facility in BSD Unix. (09 marks) (May-08/10)

Ans:

For our purposes, a race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run. The fork function is a lively breeding ground for race conditions, if any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork. In general, we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

```
static void charatatime(char *);

int
main(void)
{
   pid_t   pid;

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

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

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

Output:

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

iii)

One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, since on many workstations, the console device runs a windowing system. We also don't want each daemon writing its own error messages into a separate file. It would be a headache for anyone administering the system to keep up with which daemon writes to which log file and to check these files on a regular basis. A central daemon error-logging facility is required.

There are three ways to generate log messages:

1. Kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device. We won't describe this function any further, since we're not interested in writing kernel routines.
2. Most user processes (daemons) call the syslog(3) function to generate log messages. We describe its calling sequence later. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
3. 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.

5.Write a program to implement popen and pclose system calls. (10 marks)

Ans: (Dec-10)

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

```
/*
 * Pointer to array allocated at run-time.
 */
static pid_t    *childpid = NULL;
/*
 * From our open_max(), Figure 2.16.
 */
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 = 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 */
}
```