UNIX PROGRAMMING

MODULE-3

General fil e A PI's Files in a UNIX and POSIX system may be any one of the following types:

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

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

open

✓ This is used to establish a connection between a process and a file i.e. it is used to open an existing file for data transfer function or else it may be also be used to create a new file.

✓ The returned value of the open system call is the file descriptor (row number of the file table), which contains the i-node information.

✓ The prototype of open function is

#include #include int open(const char *pathname, int access-mode, mode_t permission);

✓ If successful, open returns a nonnegative integer representing the open file descriptor.

✓ If unsuccessful, open returns –1.

✓ The first argument is the name of the file to be created or opened. This may be an absolute pathname or relative pathname.

✓ If the given pathname is symbolic link, the open function will resolve the symbolic link reference to a non-symbolic link file to which it refers.

✓ The second argument is access modes, which is an integer value that specifies how actually the file should be accessed by the calling process.

✓ Generally the access modes are specified in . Various access modes are:

O_RDONLY - opens for reading file only.

O_WRONLY – opens for writing file only.

O_RDWR - opens for reading and writing file.

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

O_APPEND - Append data to the end of file.

O_CREAT - Create the file if it doesn't exist

O_EXCL - Generate an error if O_CREAT is also specified, and the file already exists.

O_TRUNC - If file exists discard the file content and set the file size to zero bytes.

O_NONBLOCK - Specify subsequent read or write on the file should be non-blocking.

O_NOCTTY - Specify not to use terminal device file as the calling process control terminal.

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

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

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

✓ If the file is opened in read only, then no other modifier flags can be used.

✓ If a file is opened in write only or read write, then we are allowed to use any modifier flags along with them.

✓ The third argument is used only when a new file is being created.

creat

✓ This system call is used to create new regular files.

✓ The prototype of creat is

#include <sys/types.h>

#include<unistd.h>

*int creat(const char *pathname, mode_t mode);*

✓ Returns: file descriptor opened for write-only if OK, -1 on error.

✓ The first argument pathname specifies name of the file to be created.

✓ The second argument mode_t, specifies permission of a file to be accessed by owner group and others.

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

#define creat(path_name, mode)

open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);

read

✓ The read function fetches a fixed size of block of data from a file referenced by a given file descriptor.

✓ The prototype of read function is

#include<sys/types.h>

#include<unistd.h>

*size_t read(int fdesc, void *buf, size_t nbyte);*

- ✓ If successful, read returns the number of bytes actually read.
- ✓ If unsuccessful, read returns –1.

- ✓ The first argument is an integer, fdesc that refers to an opened file.
- ✓ The second argument, buf is the address of a buffer holding any data read.
- ✓ The third argument specifies how many bytes of data are to be read from the file.
- ✓ The size_t data type is defined in the <sys/types.h> header and should be the same as unsigned int.
- ✓ There are several cases in which the number of bytes actually read is less than the amount requested:
- ✓ When reading from a regular file, if the end of file is reached before the requested number of bytes has been read.
- ✓ For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).

write

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

    #include<sys/types.h>

    #include<unistd.h>

    *ssize_t write(int fdesc, const void *buf, size_t size);*

- ✓ If successful, write returns the number of bytes actually written.
- ✓ If unsuccessful, write returns –1.

- ✓ The first argument, fdesc is an integer that refers to an opened file.
- ✓ The second argument, buf is the address of a buffer that contains data to be written.
- ✓ The third argument, size specifies how many bytes of data are in the buf argument.
- ✓ The return value is usually equal to the number of bytes of data successfully written to a file.

close

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

    #include<unistd.h>

    int close(int fdesc);

- ✓ If successful, close returns 0.
- ✓ If unsuccessful, close returns –1.

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

fcntl

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

    #include<fcntl.h>

    int fcntl(int fdesc, int cmd, …);

- ✓ The first argument is the file descriptor.
- ✓ The second argument cmd specifies what operation has to be performed.
- ✓ The third argument is dependent on the actual cmd value.

The possible cmd values are defined in <fcntl.h> header.

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

int cur_flags=fcntl(fdesc,F_GETFL);

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

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

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

(void)fcntl(fdesc,F_SETFD,1);   //turn on close-on-exec flag

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

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

int fdesc=open("FOO",O_RDONLY);   //open FOO for read

close(0);   //close standard input

if(fcntl(fdesc,F_DUPFD,0)==-1)

perror("fcntl");   //stdin from FOO now

char buf[256];

int rc=read(0,buf,256);   //read data from FOO

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

#define dup(fdesc)   fcntl(fdesc, F_DUPFD,0)

#define   dup2(fdesc1,fd2)

close(fd2),   fcntl(fdesc,F_DUPFD,fd2)

lseek

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

  #include <sys/types.h>

  #include <unistd.h>

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

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

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

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

The following is an example using the lseek system call.

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

int main()

```
{
    int file=0;
    if((file=open("testfile.txt",O_RDONLY)) < -1)
        return 1;

    char buffer[19];
    if(read(file,buffer,19) != 19)  return 1;
    printf("%s\n",buffer);

    if(lseek(file,10,SEEK_SET) < 0) return 1;

    if(read(file,buffer,19) != 19)  return 1;
    printf("%s\n",buffer);

    return 0;
}
```

The output of the preceding code is:
$ cat testfile.txt
This is a test file that will be used
to demonstrate the use of lseek.
$ ./testing
This is a test file
test file that will

link

✓ The link function creates a new link for the existing file.

✓ The prototype of the link function is

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

✓ If successful, the link function returns 0.

✓ If unsuccessful, link returns –1.

✓ The first argument cur_link, is the pathname of existing file.

✓ The second argument new_link is a new pathname to be assigned to the same file.

✓ If this call succeeds, the hard link count will be increased by 1.

The UNIX ln command is implemented using the link API.

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

unlink

- ✓ The unlink function deletes a link of an existing file.

- ✓ This function decreases the hard link count attributes of the named file, and removes the file name entry of the link from directory file.

- ✓ A file is removed from the file system when its hard link count is zero and no process has any file descriptor referencing that file.

- ✓ The prototype of unlink is

    #include <unistd.h>

    int unlink(const char * cur_link);

- ✓ If successful, the unlink function returns 0.

- ✓ If unsuccessful, unlink returns –1.

- ✓ The argument cur_link is a path name that references an existing file. ANSI C defines the rename function which does the similar unlink operation.

- ✓ The prototype of the rename function is:

    #include<stdio.h>

    int rename(const char * old_path_name,const char * new_path_name);

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

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

stat, fstat

✓ The stat and fstat function retrieves the file attributes of a given file. The only difference between stat and fstat is that the first argument of a stat is a file pathname, whereas the first argument of fstat is file descriptor.

✓ The prototypes of these functions are

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

✓ The second argument to stat and fstat is the address of a struct stat-typed variable which is defined in the

> **<sys/stat.h> header.**

Its declaration is as follows:

```
struct stat
{
dev_t   st_dev;          /* file system ID */
    ino_t    st_ino;         /* file inode number */
    mode_t  st_mode;        /* contains file type and permission */
    nlink_t  st_nlink;    /* hard link count */
    uid_t    st_uid;         /* file user ID */
```

```
gid_t    st_gid;              /* file group ID */
dev_t    st_rdev;             /*contains major and minor device#*/
off_t    st_size;             /* file size in bytes */
time_t   st_atime;            /* last access time */
time_t   st_mtime;            /* last modification time */
time_t   st_ctime;            /* last status change time */
};
```

✓ The return value of both functions is 0 if they succeed -1 if they fail *errno* contains an error status code.

✓ The lstat function prototype is the same as that of stat:

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

file type with the macros as shown

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

access

✓ The access system call checks the existence and access permission of user to a named file.

✓ The prototype of access function is

#include<unistd.h>

int access(const char *path_name, int flag);

✓ On success access returns 0, on failure it returns –1.

✓ The first argument is the pathname of a file.

✓ The second argument flag, contains one or more of the following bit flag .

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

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

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

- ✓ example to check whether a file exists:

    if(access("/usr/anjan/usp.txt", F_OK)==-1)

    printf("file does not exists");

    else

    printf("file exists");

chmod, fchmod

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

- ✓ The prototypes of these functions are

    #include<sys/types.h>

    #include<sys/stat.h>

    #include<unistd.h>

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

    int fchmod(int fdesc, mode_t flag);

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

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

chown, fchown, lchown

✓ The chown functions changes the user ID and group ID of files.

✓ The prototypes of these functions are

> *#include<unistd.h>*
>
> *#include<sys/types.h>*
>
> *int chown(const char *path_name, uid_t uid, gid_t gid);*
>
> *int fchown(int fdesc, uid_t uid, gid_t gid);*
>
> *int lchown(const char *path_name, uid_t uid, gid_t gid);*

✓ The path_name argument is the path name of a file.

✓ The uid argument specifies the new user ID to be assigned to the file.

✓ The third argument specifies the new group ID to be assigned to the file.

/ Program to illustrate chown function */*

```
#include<iostream.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
#include<pwd.h>
int main(int argc, char *argv[ ])
```

```
{
        if(argc>3)
        {
                cerr<<"usage:"<<argv[0]<<"<usr_name><file>.\n";
                return 1;
        }
        struct passwd *pwd = getpwuid(argv[1]) ;
        uid_t UID = pwd ? pwd -> pw_uid : -1 ;
        struct stat      statv;
        if (UID == (uid_t)-1)
                cerr <<"Invalid user name";
                else
                 for (int i = 2; i < argc ; i++)
        if (stat(argv[i], &statv)==0)
        {
                if (chown(argv[i], UID,statv.st_gid))
                 perror ("chown");
                else
                perror ("stat");
        }
    return 0;
    }
```

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

utime Function

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

```
#include<sys/types.h>
#include<unistd.h>
#include<utime.h>
int utime(const char *path_name, struct utimbuf *times);
```

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

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

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

File and Record Locking

- ✓ Multiple processes perform read and write operation on the same file concurrently. This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism.
- ✓ File locking is applicable for regular files. Only a process can impose a write lock or read lock on either a portion of a file or on the entire file.
- ✓ The differences between the read lock and the write lock is that when write lock is set, it prevents the other process from setting any over-lapping read or write lock on the locked file. Similarly, when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region.
- ✓ The intension of the write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region, so write lock is termed as "**Exclusive lock**".

✓ The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region. Other processes are allowed to lock and read data from the locked regions. Hence a read lock is also called as "**shared lock** ".

✓ File lock may be **mandatory** if they are enforced by an operating system kernel. If a mandatory exclusive lock is set on a file, no process can use the read or write system calls to access the data on the locked region. These mechanisms can be used to synchronize reading and writing of shared files by multiple processes.

✓ If a process locks up a file, other processes that attempt to write to the locked regions are blocked until the former process releases its lock.

✓ Problem with mandatory lock is – if a runaway process sets a mandatory exclusive lock on a file and never unlocks it, then, no other process can access the locked region of the file until the runway process is killed or the system has to be rebooted.

✓ If locks are not mandatory, then it has to be **advisory** lock. A kernel at the system call level does not enforce advisory locks. This means that even though a lock may be set on a file, no other processes can still use the read and write functions to access the file.

✓ To make use of advisory locks, process that manipulate the same file must co-operate such that they follow the given below procedure for every read or write operation to the file. Try to set a lock at the region to be accesses. If this fails, a process can either wait for the lock request to become successful. After a lock is acquired successfully, read or write the locked region.

Release the lock.

✓ If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512, the previous read lock from 0 to 256 is now covered by the write lock and the process does not own two locks on the region from 0 to 256. This process is called "**Lock Promotion**".

✓ If a process now unblocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called "**Lock**

Splitting".

✓ UNIX systems provide fcntl function to support file locking. By using fcntl it is possible to impose read or write locks on either a region or an entire file.

✓ The prototype of fcntl is

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

✓ The first argument specifies the file descriptor.

✓ The second argument cmd_flag specifies what operation has to be performed.

✓ If fcntl is used for file locking then it can values as

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

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

struct flock

{

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

};

✓ The l_type field specifies the lock type to be set or unset.

✓ The possible values, which are defined in the <fcntl.h> header, and their   uses are

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

The l_whence, l_start & l_len define a region of a file to be locked or unlocked.

The possible values of l_whence and their uses are

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

✓ A lock set by the fcntl API is an advisory lock but we can also use fcntl for mandatory locking purpose with the following attributes set before using fcntl Turn on the set-GID flag of the file. Turn off the group execute right permission of the file.

Consider the last 100 bytes as a region. Write a C/C++ program to check whether the region is locked or not. If the region is locked, print pid of the process which has locked. If the region is not locked, lock the region with an exclusive lock, read the last 50 bytes and unlock the region.

- **server.c** : is used to lock a region and after locking print process id of process which has been locked and unlock the region

- **client.c** : is used to find whether it has been locked by server and to display which type of lock ,if there is no lock, lock the region with exclusive lock, read the last 50 bytes and unlock the region

server.c
```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
#include<errno.h>
#include<unistd.h>
int main()
{
	struct  flock f1;
	int fd;
	//initializing locking parameters
	f1.l_pid  = getpid();
	f1.l_type = F_WRLCK; // write lock
	f1.l_whence = SEEK_SET; // from beginning
	f1.l_start = 0;
	f1.l_len = 0; // to the end of the file
	fd= open("k1",O_WRONLY);
	// apply lock
	if(fcntl(fd,F_SETLK,&f1)==-1)
	{
		perror("fcntl");
		exit(1);
	}
	getchar();
	printf("got lock");
	printf("pid of the process %ld",getpid());
	getchar();
	f1.l_type =F_UNLCK; //set to unlock the same region
```

```
        if(fcntl(fd,F_SETLK,&f1) == -1)
        {
                perror("fcntl");
                exit(1);
        }
        printf("Unlocked\n");
        close(fd);
        return 0;
}
```

client.c

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
#include<errno.h>
#include<unistd.h>
int main()
{
        struct flock f1={ F_WRLCK, SEEK_END ,0,100,0);
        int fd;
        char a;
        fd= open("k1",O_RDWR,0);
        // check whether it has been locked
        if(fcntl(fd,F_GETLK,&f1) == -1)
        {
        perror("fcntl");
        exit(1);
        }       // if it is locked display
        if(f1.l_type == F_RDLCK)
                printf("\n read lock");
        if(f1.l_type == F_WRLCK)
                printf("\n Write lock");
        printf("pid process= %ld", f1.l_pid);
        if(f1.l_type == F_UNLCK)
        {
                printf("\n no lock");
                // if it is not locked –lock the file
                fcntl(fd,F_SETLK,&f1);
                lseek(fd, -50,SEEK_END);
                printf("contents of file is \n");          // display the contents
        while(read(fd,&a,1)>0)
        printf("%c",a);
        f1.l_type = F_UNLCK;
        fcntl( fd, F_SETLK, &f1);
  }
        return 0;
}
```

Directory File APIs

- ✓ Directory files are used in organizing files, based on the specific use of the file.
- ✓ Directory files are also used by OS to convert file path to their inode numbers.

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

- ➢ mode argument specifies, access permission for owner, group and others to be assigned to DF.
- ➢ mode value is modified by the calling process **umask**.
- ✓ Return value of mkdir is 0 on success, -1 if it fails.
- ✓ Possible cause of failure

> - path_name is invalid
>
> - calling process lacks permission to create specified directory.
>
> - mode argument is invalid.

- ✓ A directory created by mkdir has "." and ".." links created in one **atomic operation**.

"." refers to pwd,   ".." refers to the parent directory of pwd.

- ✓ Newly created directory has its UID set to the EUID of the process that creates it. Directory GID will be set to either the EGID of calling process or GID of parent directory that hosts the new directory.
- ✓ A directory file is a record-oriented file, where each record stores a file name and the inode number of a file that resides in that directory.
- ✓ Directory record structure differs based on different unix flavours. Following portable functions are defined for directory file.

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

Dirent   **readdir** (**Dir*** dir_fdesc);*
*int   **closedir** (**Dir *** dir_fdesc);*
*void   **rewinddir** (**Dir *** dir_fdesc);*

➢ opendir - Opens a directory file for **read-only**, returns a file handle DIR* for future reference of the file.

➢ readdir - Reads a record from a directory file referenced by dir_fdesc and returns that record information.

➢ closedir - Closes a directory file referenced by dir_fdesc.

➢ rewinddir - Resets file pointer to the beginning of the directory file referenced by dir_fdesc. Next call to

✓ readdir will read the first record from the file. When readdir is called after the opendir or rewinddir API, it will return the first data record from the file, on the next call it will return the second data record in the file etc., when readdir has scanned all records in a directory file, it will return a zero value to indicate that end-of-file has been reached.

✓ UNIX systems have defined additional functions for random access of directory file records.

     ➢ telldir   Returns the file pointer of a give dir_fdesc.

     ➢ Seekdir   Changes the file pointer of a give dir_fdesc to a specified address.

✓ Directory files are removed by **rmdir** API. This requires that the directories to be removed be empty, in that they contain no files other than the "."and ".." links.

       #include <unistd.h>

       int   **rmdir** (**const char *** path_name);

Device File APIs

✓ Device files are used to interface physical devices with application program. Device files are either **character-based** or **block-based**.

✓ When a process reads from or writes to a device file, the file's **major device number** is used to locate and invoke a device driver function that does the actual data transmission with the physical device.

✓ **Minor device number** is an argument being passed to the device driver function when

it is invoked.

✓ This is needed because a device driver function may be used for different types of device, and the minor device number specifies the parameters (like buffer size) to be used for a particular device type.

    #include <sys/stat.h>

    #include <unistd.h>

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

➤ path_name is the path name of a device file to be created.

➤ mode specifies the access permission, for the owner, group and others to be assigned to the file, as well as S_IFCHR / S_IFBLK flags, these are used to indicate whether the file being created is a character or block device file. Access permission is modified by the calling process umask.

➤ device_id contains the major and minor device numbers, which is constructed as follows. The lowest byte of a device_id is set to a minor device number and the next byte is set to the major device number.

✓ Ex: to create a block device file called SCSI5 with major and minor number of 15 and 3 and access permission of r, w and x for everyone

mknod("SCSI5", S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO, (15<<8) | 3);

✓ mknod API must be called by a process with superuser privileges. UID and GID attributes of a device file are assigned in the same manner as for regular file. File size attribute of any device file has no usage.

✓ Return value of mknod is 0 if it succeeds or -1 if it fails. Possible failure includes

            - path_name specified is invalid

            - process lacks permission to create a device file

            - mode argument is invalid.

✓ Once a device file is created, any process may use open API to connect to the file. It can use read, write, stat and close API to manipulate the file. lseek is applicable to block device files only.

- ✓ Device files can be removed using unlink API.
- ✓ Only privileged users (e.g. superuser) can use mknod API to create a device files. All other users may read and write device files as if they were regular files, subjected to the access permissions set on those device files.

FIFO File APIs

Ex: mkfifo("FIFO5", S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO);

- ✓ Return value of mkfifo is 0 if it succeeds or -1 if it fails. Possible failures may be

  - path name specified is invalid

  - a process lacks permission to create the file

  - an invalid mode argument is specified.

- ✓ Once a FIFO file is created any process may use the open API to connect to the file. Process can use read, write, stat and close APIs to manipulate the file. **lseek is not applicable** to FIFO files. FIFO file may be removed via unlik API.
- ✓ When a process opens a FIFO file for read-only, kernel will block the process until there is another process that opens the same file for write. Similarly, if a process opens a FIFO for write, it will be blocked until another process opens the FIFO file for read.
- ✓ If a process write to a FIFO that is full, the process will be blocked until another process has read data from the FIFO to make room for new data in FIFO. Conversely, if a process attempts to read data from a FIFO that is empty, the process will be blocked until another process writes data to the FIFO.
- ✓ If a process uses O_NONBLOCK flag in the open call to FIFO file, the **process will not be blocked** even though there is no process attached to the other end of the FIFO file. Further, if the process subsequently calls the read or write API on FIFO file and data is not ready for transfer, these functions will return immediately with a -1 value and set error to EAGAIN. Thus the process continues to do some other task and try these operations after some time.

Symbolic Link File API's

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

The prototype is

 #include<unistd.h>

#include <sys/types.h>

#include <sys/stst.h>

int symlink(const char *org_link, const char *sym_link);

int readlink(const char* sym_link,char* buf,int size);

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

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

UNIX PROCESSES

The Environment of a UNIX Process

INTRODUCTION

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

mainFUNCTION

- ✓ 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 mainfunction is called.

PROCESS TERMINATION

- ✓ There are eight ways for a process to terminate. Normal termination occurs in five ways:
    - ➢ Return from main
    - ➢ Calling exit
    - ➢ Calling _exitor _Exit
    - ➢ Return of the last thread from its start routine
    - ➢ Calling pthread_exit from the last thread

- ✓ Abnormal termination occurs in three ways:

> ➢ Calling abort
>
> ➢ Receipt of a signal
>
> ➢ Response of the last thread to a cancellation request

Exit Functions

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

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

✓ All three exit functions expect a single integer argument, called the exit status. Returning an integer value from the main function is equivalent to calling exit with the same value. **exit(0);**

✓ is the same as **return(0);**

✓ from the main function.

atexit Function

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

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

✓ Returns: 0 if OK, nonzero on error

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

The below figure summarizes how a C program is started and the various ways it can terminate
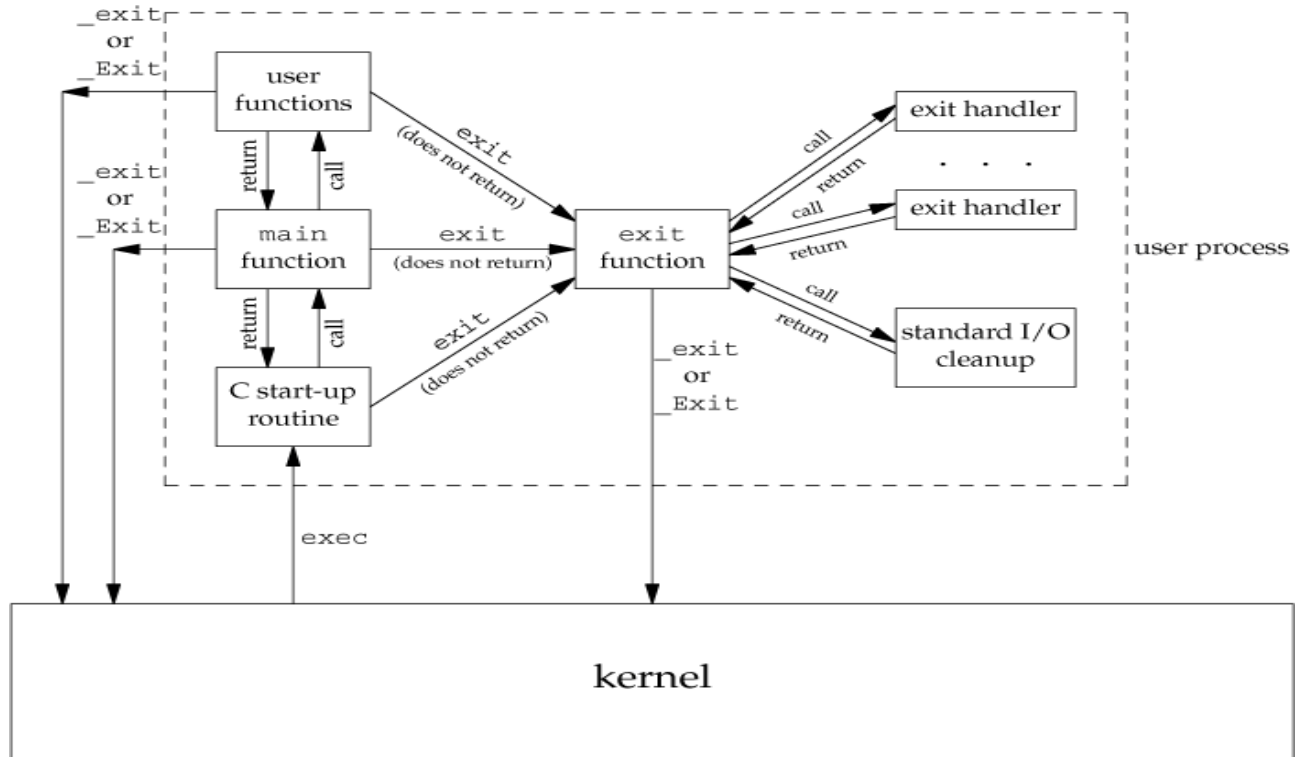
Example of exit handlers

```
static void my_exit1(void);
static void my_exit2(void);
int main(void)
{
        if (atexit(my_exit2) != 0)
                err_sys("can't registermy_exit2");
        if (atexit(my_exit1) != 0)
                err_sys("can't registermy_exit1");
        if (atexit(my_exit1) != 0)
                err_sys("can't registermy_exit1");
                printf("main is done\n"); return(0);
}
static void
        my_exit1(void)
        {
                printf("first exit handler\n");
        }
static void
        my_exit2(void)
        {
```

printf("second exit handler\n");

}

Output:

$ ./a.out

main is done

first exit handler

first exit handler

second exit handler

COMMAND-LINE ARGUMENTS

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

- ✓ Example: Echo all command-line arguments to standard output

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

Output:

$ ./echoarg arg1 TEST foo

argv[0]: ./echoarg

argv[1]: arg1

argv[2]: TEST

argv[3]: foo

ENVIRONMENT LIST

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

    extern char **environ;

Figure: Environment consisting of five C character strings
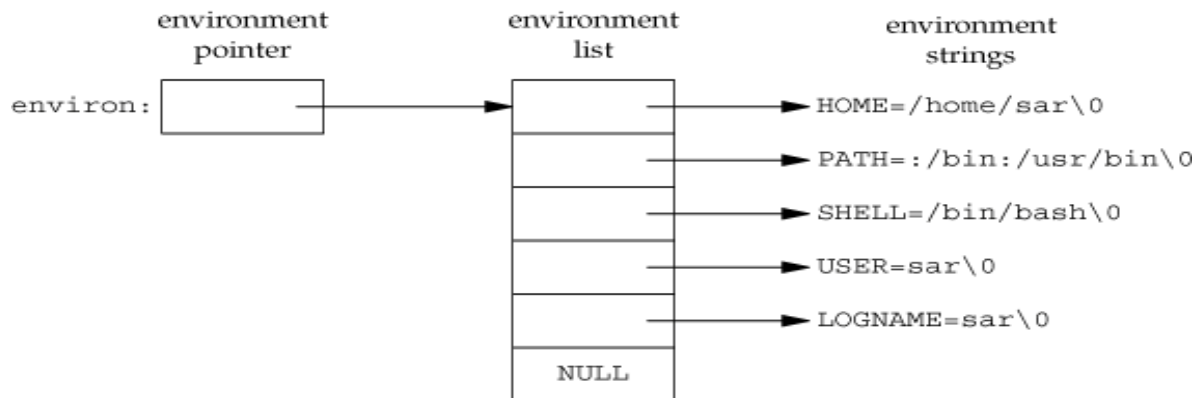
- ✓ Generally, any environmental variable is of the form: ***name=value.***

MEMORY LAYOUT OF A C PROGRAM

- ✓ C program has been composed of the following :

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

- ❖ **Initialized data segment**, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration

  int   maxcount = 99;

  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

- ❖ **Uninitialized data segment**, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration

  long   sum[1000];

  appearing outside any function causes this variable to be stored in the uninitialized data segment.

❖ **Stack**, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

❖ **Heap**, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.
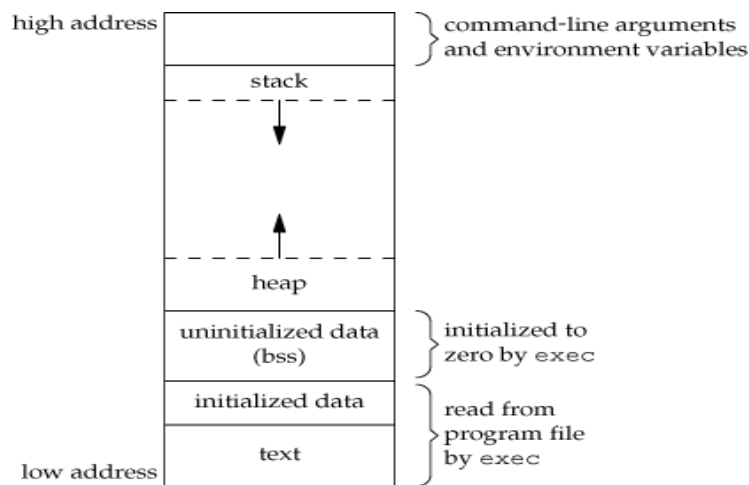
SHARED LIBRARIES

✓ Most UNIX systems support shared libraries. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference. This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called.

✓ Another advantage of shared libraries is that, library functions can be replaced with new versions without having to re-link, edit every program that uses the library. With cc compiler we can use the option –g to indicate that we are using shared library.

MEMORY ALLOCATION

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

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

- ✓ All three return: non-null pointer if OK, NULLon error

> **void free(void *ptr);**

- ✓ The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. Because the three alloc functions return a generic void * pointer, if we #include <stdlib.h> (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type.
- ✓ The function free causes the space pointed to by ptr to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three alloc functions.
- ✓ The realloc function lets us increase or decrease the size of a previously allocated area. For example, if we allocate room for 512 elements in an array that we fill in at runtime but find that we need room for more than 512 elements, we can call realloc. If there is room beyond the end of the existing region for the requested space, then realloc doesn't have to move anything; it simply allocates the additional area at the end and returns the same pointer that we passed it. But if there isn't room at the end of the existing region,

realloc allocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area.

ENVIRONMENT VARIABLES

✓ The environment strings are usually of the form: *name=value.* The UNIX kernel never looks at these strings; their interpretation is up to the various applications. The shells, for example, use numerous environment variables. Some, such as HOME and USER, are set automatically at login, and others are for us to set.

✓ Set environment variables in a shell start-up file to control the shell's actions. The functions that we can use to set and fetch values from the variables are setenv, putenv, and getenv functions.

✓ The prototype of these functions are

> *#include <stdlib.h>*

> *char *getenv(const char *name);*

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

✓ This function returns a pointer to the value of a **name=value** string.

✓ We should always use getenv to fetch a specific value from the environment, instead of accessing environ directly. In addition to fetching the value of an environment variable.

✓ The prototypes of these functions are

> *#include <stdlib.h>*

> *int putenv(char *str);*

> *int setenv(const char *name, const char *value, int rewrite);*

> *int unsetenv(const char *name);*

✓ All return: 0 if OK, nonzero on error.

✓ The **putenv** function takes a string of the form **name=value** and places it in the environment list. If name already exists, its old definition is first removed.

- ✓ The **setenv** function sets name to value. If name already exists in the environment, then if rewrite is nonzero, the existing definition for name is first removed.If rewrite is 0, an existing definition for name is not removed, name is not set to the new value, and no error occurs.

- ✓ The **unsetenv** function removes any definition of name. It is not an error if such a definition does not exist.

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

Environment variables defined in the Single UNIX Specification

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

setjmp AND longjmp FUNCTIONS

5) What is the use of setjmp and longjmp functions? Write a C program to illustrate the same.

- ✓ The setjmp and longjmp functions were used to perform branching.

  #include <setjmp.h>

  int setjmp(jmp_buf env);

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

  void longjmp(jmp_buf env, int val);

- ✓ The setjmp function records or marks a location in a program code so that later when the longjmp function is called from some other function, the execution continues from the location onwards.

- ✓ The env variable(the first argument) records the necessary information needed to continue execution. The env is of the jmp_buf defined in <setjmp.h> file, it contains the task.

Example of setjmpand longjmp

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

- ✓ The setjmp function always returns '0' on its success when it is called directly in a process (for the first time).

✓ The longjmp function is called to transfer a program flow to a location that was stored in the env argument.

✓ The program code marked by the env must be in a function that is among the callers of the current function.

✓ When the process is jumping to the target function, all the stack space used in the current function and its callers, upto the target function are discarded by the longjmp function.

✓ The process resumes execution by re-executing the setjmp statement in the target function that is marked by env. The return value of setjmp function is the value(val), as specified in the longjmp function call. The 'val' should be nonzero, so that it can be used to indicate where and why the longjmp function was invoked and process can do error handling accordingly.

getrlimit AND setrlimit FUNCTIONS

✓ Every process has a set of resource limits, some of which can be queried and changed by the gerllimit and setrlimitfunctions

> **#include <sys/resource.h>**
>
> **int getrlimit(int resource, struct rlimit *rlptr);**
>
> **int setrlimit(int resource, const struct rlimit   *rlptr);**

✓ Both return: 0 if OK, nonzero on error

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

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

RLIMIT_AS	The maximum size in bytes of a process's total available memory.
RLIMIT_CORE	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
RLIMIT_CPU	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.
RLIMIT_DATA	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.

RLIMIT_FSIZE	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZsignal.
RLIMIT_LOCKS	The maximum number of file locks a process can hold.
RLIMIT_MEMLOCK	The maximum amount of memory in bytes that a process can lock into memory using mlock(2).
RLIMIT_NOFILE	The maximum number of open files per process. Changing this limit affects the value returned by the sysconffunction for its _SC_OPEN_MAXargument
RLIMIT_NPROC	The maximum number of child processes per real user ID. Changing this limit affects the value returned for _SC_CHILD_MAXby the sysconffunction
RLIMIT_RSS	Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.
RLIMIT_SBSIZE	The maximum size in bytes of socket buffers that a user can consume at any given time.
RLIMIT_STACK	The maximum size in bytes of the stack.
RLIMIT_VMEM	This is a synonym for RLIMIT_AS.

- ✓ Three rules govern the changing of the resource limits.
    - o A process can change its soft limit to a value less than or equal to its hard limit.
    - o A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users. Only a superuser process can raise a hard limit.
- ✓ An infinite limit is specified by the constant RLIM_INFINITY.
- ✓ The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes.

UNIX KERNEL SUPPORT FOR PROCESS

- ✓ The data structure and execution of processes are dependent on operating system implementation. A UNIX process consists minimally of a text segment, a data segment and a stack segment.
- ✓ A segment is an area of memory that is managed by the system as a unit.
- ✓ A text segment consists of the program text in machine executable instruction code

format.

✓ The data segment contains static and global variables and their corresponding data.

✓ A stack segment contains runtime variables and the return addresses of all active functions for a process.

✓ UNIX kernel has a process table that keeps track of all active process present in the system. Some of these processes belongs to the kernel and are called as "system process".

✓ Every entry in the process table contains pointers to the text, data and the stack segments and also to U-area of a process.

✓ U-area of a process is an extension of the process table entry and contains other process specific data such as the file descriptor table, current root and working directory inode numbers and set of system-imposed process limits.

✓ When a process is created by fork, it contains duplicated copies of the text, data and stack segments of its parent as shown in the Figure below.

✓ It has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

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

✓ **A real user identification number (rUID)**: the user ID of a user who created the parent process.

✓ **A real group identification number (rGID)**: the group ID of a user who created that parent process.

✓ **An effective user identification number (eUID)**: this allows the process to access and create files with the same privileges as the program file owner.

✓ **An effective group identification number (eGID)**: this allows the process to access and create files with the same privileges as the group to which the program file belongs.

✓ **Saved set-UID and saved set-GID**: these are the assigned eUID and eGID of the process respectively.

A Unix process data structure
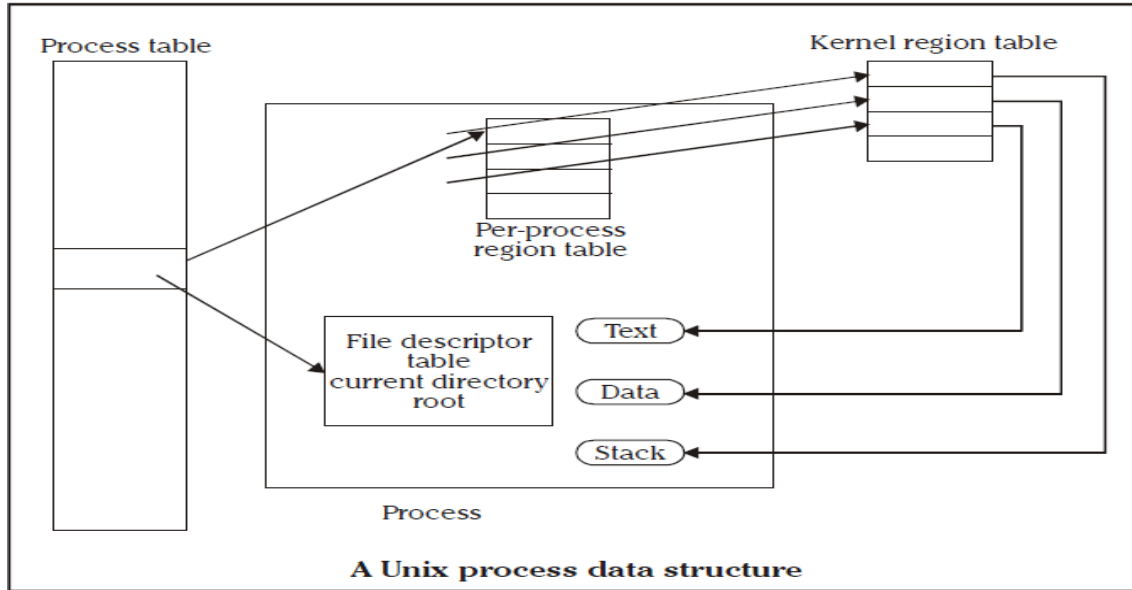
✓ All processes in UNIX system expect the process that is created by the system boot code, are created by the fork system call. After the fork system call, once the child process is created, both the parent and child processes resumes execution.
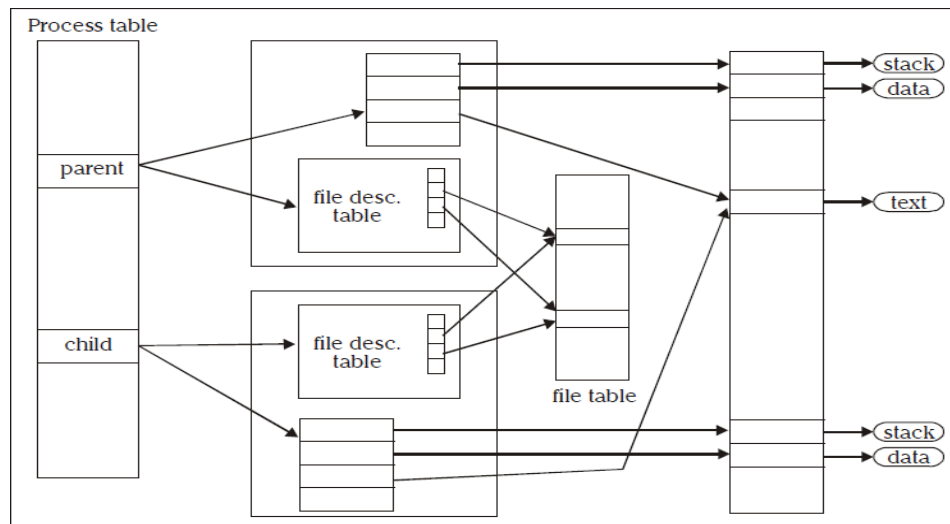
✓ **Process group identification number (PGID) and session identification number (SID):** these identify the process group and session of which the process is member.

✓ **Supplementary group identification numbers:** this is a set of additional group IDs for a user who created the process.

✓ **Current directory:** this is the reference (inode number) to a working directory file.

✓ **Root directory**: this is the reference to a root directory.

- ✓ **Signal handling**: the signal handling settings.
- ✓ **Signal mask**: a signal mask that specifies which signals are to be blocked.
- ✓ **Unmask**: a file mode mask that is used in creation of files to specify which accession rights should be taken out.
- ✓ **Nice value**: the process scheduling priority value.
- ✓ **Controlling terminal**: the controlling terminal of the process.

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

- ✓ **Process identification number (PID)**: an integer identification number that is unique per process in an entire operating system.
- ✓ **Parent process identification number (PPID)**: the parent process PID.
- ✓ **Pending signals**: the set of signals that are pending delivery to the parent process.
- ✓ **Alarm clock time**: the process alarm clock time is reset to zero in the child process.
- ✓ **File locks**: the set of file locks owned by the parent process is not inherited by the chid process.
- ✓ *fork* and *exec* are commonly used together to spawn a sub-process to execute a different program. The advantages of this method are:

A process can create multiple processes to execute multiple programs concurrently.

Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of its child process.

PROCESS CONTROL

PROCESS IDENTIFIERS

#include <unistd.h>

pid_t getpid(void);

❖ Returns: process ID of calling process

pid_t getppid(void);

❖ Returns: parent process ID of calling process

uid_t getuid(void);

❖ Returns: real user ID of calling process

uid_t geteuid(void);

❖ Returns: effective user ID of calling process

gid_t getgid(void);

❖ Returns: real group ID of calling process

gid_t getegid(void);

❖ Returns: effective group ID of calling process

fork FUNCTION

6) Explain fork( ) and vfork( ) system call. Using a C program illustrate how fork system call differs from vfork.

✓ An existing process can create a new one by calling the forkfunction.

#include <unistd.h>

pid_t fork(void);

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

✓ The new process created by forkis called the child process.

✓ This function is called once but returns twice.

✓ The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.

✓ The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.

✓ The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

✓ Both the child and the parent continue executing with the instruction that follows the call to fork.

✓ The child is a copy of the parent.

✓ For example, the child gets a copy of the parent's data space, heap, and stack.

✓ Note that this is a copy for the child; the parent and the child do not share these portions of memory.

✓ The parent and the child share the text segment .

Example programs:

Program 1
/ Program to demonstrate fork function Program name – fork1.c */*

```
int main()
{
        fork( );
        printf("\n hello USP");
}
```

Output :
$ cc fork1.c
$ ./a.out
hello USP
hello USP

Program 2
/ Program name – fork2.c */*

```
int main( )
{
        printf("\n 6 sem ");
        fork( );
        printf("\n hello USP");
}
```

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

vfork FUNCTION

- ✓ The function vforkhas the same calling sequence and same return values as fork.
- ✓ The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.
- ✓ The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space, the child simply calls exec (or exit) right after the vfork. Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System. Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes.

Example of vforkfunction

```
int      glob = 6;        /* external variable in initialized data */
int main(void)
{
        int      var;    /* automatic variable on the stack */
        pid_t    pid;
        var = 88;
        printf("before vfork\n");        /* we don't flush stdio */
         if ((pid = vfork()) < 0)
        {
                err_sys("vfork error");
        }
         Else
         if (pid == 0)
         {       /* child */
`               glob++;        /* modify parent's variables */ var++;
_               exit(0);/* child terminates */
         }
    /*
    * Parent continues here.
    */
        printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var); exit(0);
}
```

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

exit FUNCTIONS

- ✓ A process can terminate normally in five ways:
    - o Executing a return from the main function.
    - o Calling the exit function.
    - o Calling the _exit or _Exit function.
- ✓ In most UNIX system implementations, exit(3) is a function in the standard C library, whereas _exit(2) is a system call.
- ✓ Executing a return from the start routine of the last thread in the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
- ✓ Calling the pthread_exit function from the last thread in the process. The three forms of abnormal termination are as follows:
- ✓ Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.
- ✓ When the process receives certain signals. Examples of signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.
- ✓ The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later, the target thread terminates.

8)   Discuss waitpid() function with sutable example

wait AND waitpid FUNCTIONS

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

✓ A process that calls wait or waitpid can: Block, if all of its children are still running Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched Return immediately with an error, if it doesn't have any child processes.

1) What is a Zombie process?

> **#include <sys/wait.h>**
>
> **pid_t wait(int *statloc);**
>
> **pid_t waitpid(pid_t pid,**
>
> **int *statloc, int options);**

✓ Both return: process ID if OK, or 1 on error.

✓ The differences between these two functions are as follows.

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

- o The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

✓ If a child has already terminated and is a <mark>zombie,</mark> wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates. For both functions, the argument statloc is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

17) Write a program to demonstrate Zombie process?

Program to Demonstrate various exitstatuses

3) Write a C program to avoid Zombie process by forking twice.

```
int main(void)
{
        pid_t   pid;
        int      status;
        if ((pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)      /* child */
                exit(7);
        if (wait(&status) != pid)       /* wait for child */
                err_sys("wait error");
                pr_exit(status);        /* and print its status */
        if ((pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)      /* child */
```

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

The interpretation of the pid argument for waitpiddepends on its value:

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

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

The options constants for waitpid	
Constant	**Description**
WCONTINUED	If the implementation supports job control, the status of any child specified by

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

- ✓ The waitpidfunction provides three features that aren't provided by the waitfunction.

- ✓ The waitpid function lets us wait for one particular process, whereas the wait function returns the status of any terminated child. We'll return to this feature when we discuss the popenfunction.

- ✓ The waitpid function provides a nonblocking version of wait. There are times when we want to fetch a child's status, but we don't want to block.

- ✓ The waitpid function provides support for job control with the WUNTRACED and WCONTINUEDoptions.

Program to Avoid zombie processes by calling forktwice

```
int main(void)
{
        pid_t   pid;
        if ((pid = fork()) < 0)
        {
                err_sys("fork error");
        }
         else
        if (pid == 0)
        {        /* first child */
        if ((pid = fork()) < 0)
                err_sys("fork error");
                 else
                if (pid > 0)
                        exit(0);/* parent from second fork == first child */
```

/   We're the second child; our parent becomes init as soon as our real parent calls exit() in the statement above. Here's where we'd continue executing, knowing that when we're done, init will reap our status.   */*

Sleep();

printf("second child, parent pid = %d\n", getppid()); exit(0);

}

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

err_sys("waitpid error");

*/**

We're the parent (the original process); we continue executing, knowing that we're not the parent of the second child.

**/*

Exit(0);

}

Output:

$ ./a.out

$ second child, parent pid = 1

waitid FUNCTION

✓ The waitidfunction is similar to waitpid, but provides extra flexibility.

#include <sys/wait.h>

Int waited(idtype_t idtype, id_t id, siginfo_t *infop, intoptions);

✓ Returns: 0 if OK, -1 on error

The *idtype* constants for waited are as follows:

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

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

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

wait3 AND wait4 FUNCTIONS

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

- ✓ The prototypes of these functions are:

  #include <sys/types.h>

  #include <sys/wait.h>

  #include <sys/time.h>

  #include <sys/resource.h>

  pid_t wait3(int *statloc, int options, struct rusage*rusage);

  pid_t wait4(pid_t pid, int *statloc, int options, struct rusage*rusage);

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

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

RACE CONDITIONS

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

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

```
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

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

```
static void charatatime(char *);
int main(void)
{
        pid_t   pid;
        TELL_WAIT();
        if ((pid = fork()) < 0)
        {
                err_sys("fork error");
        }
         else

                if (pid == 0)
                {
                        WAIT_PARENT();      /* parent goes first */
                        charatatime("output from child\n");
                }
                else

                        {
                                charatatime("output from parent\n");
                                TELL_CHILD(pid);

                        }
        exit(0);
}
static void charatatime(char *str)
{
        char    *ptr;
        int     c;
        setbuf(stdout, NULL); /* set unbuffered */
        for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

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

exec FUNCTIONS

✓ When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely

replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk.

✓ There are 6 exec functions:

#include <unistd.h>

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

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

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

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

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

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

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

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

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

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

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

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

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

✓ The process ID does not change after an exec, but the new program inherits additional properties from the calling process:
  o Process ID and parent process ID
  o Real user ID and real group ID
  o Supplementary group IDs
  o Process group ID
  o Session ID
  o Controlling terminal
  o Time left until alarm clock
  o Current working directory
  o Root directory
  o File mode creation mask
  o File locks
  o Process signal mask
  o Pending signals
  o Resource limits
  o Values for tms_utime, tms_stime, tms_cutime, and tms_cstime.
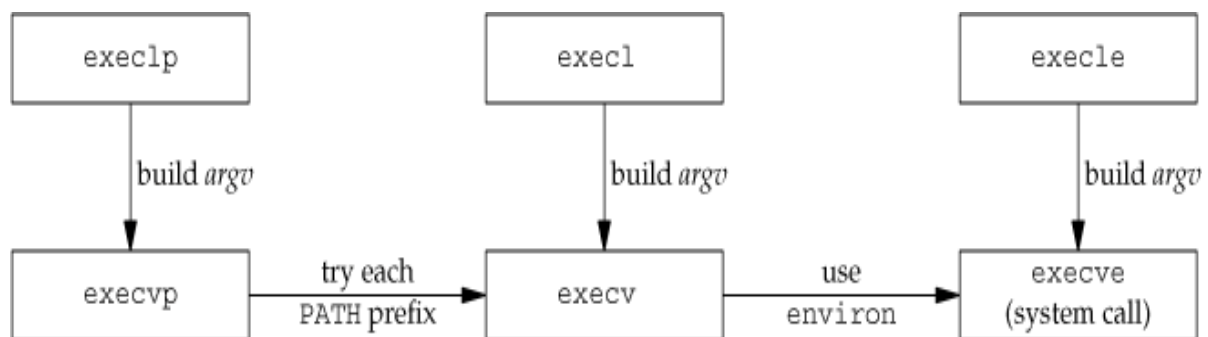
Relationship of the six exec functions

Example of execfunctions

```
#include "apue.h"
#include <sys/wait.h>
char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main(void)
{
        pid_t   pid;
        if ((pid = fork()) < 0)
        {
                err_sys("fork error");
        }
        Else
         if (pid == 0)
        {                  /* specify pathname, specify environment */
                 if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                 "MY ARG2", (char *)0, env_init) < 0) err_sys("execle error");
        }
                if (waitpid(pid, NULL, 0) < 0)
                        err_sys("wait error");
                if ((pid = fork()) < 0)
                 {
                        err_sys("fork error");
                }
         else
        if (pid == 0)
        { /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
                err_sys("execlp error");
        }
        exit(0);
}
```

Output:

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

USER=sar
LOGNAME=sar
SHELL=/bin/bash
HOME=/home/sar… etc

✓ shell prompt appeared before the printing of argv[0] from the second exec. This is because the parent did not wait for this child process to finish. Echo all command-line arguments and all environment strings

```
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);
}
```