Module-3

Table of contents

Chapter 1

UNIX FILE API'S

This chapter describes how the UNIX applications interface with files. After reading this chapter, students should be able to write programs that performs the following functions or any type of files in a UNIX system.

- Create files.
- Open files.
- Transfer data to and from files.
- Close files.
- Remove files.
- Query file attributes.
- Change file attributes.
- Truncate files.

To illustrate the applications of UNIX API's for files, some C++ programs are depicted to show the implementation of the UNIX commands ls, mv, chmod, chown and touch based on these API'S. This chapter defines a C++ class called *file*. This *file* class inherits all the properties of the C++ *fstream* class and it has additional member functions to create objects of any file type as well as to display and change file object attributes.

1.1 General File APIs

File in UNIX system may be one of the following types.

- Regular file.
- Directory file.
- FIFO file.
- Character device file.
- Block device file.
- Symbolic Link file.

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

FILE APIs USE

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

1.1.1 Open:

It is used to open or create a file by establishing a connection between the calling process and a file. It can be used to create brand new files and after file is created any process can call the *open* function to get a file descriptor to refer to the file. The file descriptor is used in the *read* and *writes* system calls to access the file content.

The prototype of the *open* function is:

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

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

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

Access mode fl**ags:**

O_RDONLY	Open the file for read only.
O_WRONLY	Open the file for write only
O_RDWR	Open the file for read and write

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

Access Modifier Flags:

O_APPEND	Appends data to the end of the file.
O_CREAT	Create the file if it does not exist.
O_EXCL	Used with O_CREAT, if the file exists, the call fails. The test for existence and the creation if the file does not exists.
O_TRUNC	If the file exits, discards the file contents and sets the file size to zero.
O_NOCTTY	Species not to use the named terminal device file as the calling process control terminal.
O_NONBLOCK	Specifies that any subsequent read or write on the file should be non-blocking.

Example, a process is normally blocked on reading an empty pipe or on writing to a pipe that is full. It may be used to specify that such read and write operations are non-blocking.

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

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

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

Permission:

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

For example 0764 specifies 7 is for owner, 6 is for group and 4 is for other.

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

6 = 110 specifies read, write permission for group.

4 = 100 specifies read permission for others.

Each bit is either 1, which means a right is granted or zero, for no such rights.

- POSIX.1 defines the permission data type as mode_t and its value is manifested constants which are aliases to octal integer values. For example, 0764 permission value should be specified as:

S_IRWXU|S_IRGRP|S_IWGRP|S_IROTH

- *Permission* value is modified by its calling process *umask* value. An *umask* value specifies some access rights to be masked off (or taken away) automatically on any files created by process.

The function prototype of the *umask* API is:

```
mode_t umask (mode_t new_umask);
```

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

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

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

1.1.2 creat:

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

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

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

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

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

#define creat (path_name, mode)

open(path_name, O_WRONLY|O_CREAT|O_TRUNC, mode)

1.1.3 read:

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

Its prototype is:

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

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

****Note:** read function can read text or binary files. This is why the data type of buf is a universal pointer (void *).

For example the following code reads, sequentially one or more record of struct sample-typed data from a file called dbase:

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

- The return value of *read* is the number of bytes of data successfully read and stored in the *buf* argument. It should be equal to the *size* value.
- If a file contains less than *size* bytes of data remaining to be read, the return value of *read* will be less than that of *size*. If end-of-file is reached, *read* will return a zero value.
- *size_t* is defined as *int* in <sys/types.h> header, users should not set *size* to exceed INT_MAX in any *read* function call.
- If a read function call is interrupted by a caught signal and the OS does not restart the system call automatically, POSIX.1 allows two possible behaviors:

1. The read function will return -1 value, errno will be set to EINTR, and all the data will be discarded.
2. The read function will return the number of bytes of data read prior to the signal interruption. This allows a process to continue reading the file.

The read function may block a calling process execution if it is reading a FIFO or device file and data is not yet available to satisfy the read request. Users may specify the O_NONBLOCK or O_NDELAY flags on a file descriptor to request nonblocking read operations on the corresponding file.

1.1.4 write:

The write function puts a fixed size block of data to a file referenced by a file descriptor
Its prototype is:

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

fdesc: is an integer file descriptor that refers to an opened file.
buf: is the address of a buffer which contains data to be written to the file.
size: specifies how many bytes of data are in the buf argument.
****Note:** write function can read text or binary files. This is why the data type of buf is a universal pointer (void *). For example, the following code fragment writes ten records of struct sample-types data to a file called dbase2:

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

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

1.1.5 close:

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

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

fdesc: is an integer file descriptor that refers to an opened file.

- The return value of close is zero if the call succeeds or -1 if it fails.
- The close function frees unused file descriptors so that they can be reused to reference other files.
- The close function will deallocate system resources which reduces the memory requirement of a process.
- If a process terminates without closing all the files it has opened, the kernel will close files for the process.

1.1.6 fcntl:

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

Its prototype is:

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

fdesc: is an integer file descriptor that refers to an opened file.

cmd: specifies which operation to perform on a file referenced by the fdesc argument.

The third argument value, which may be specified after cmd is dependent on the actual cmd value.

The possible cmd values are defined in the <fcntl.h> header. These values and their uses are:

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

The fcntl function is useful in changing the access control flag of a file descriptor.

For example: After a file is opened for blocking read-write access and the process needs to change the access to nonblocking and in write-append mode, it can call fcntl on the file's descriptor as:

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

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

1.1.7 lseek:

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

Its prototype is:

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

fdesc: is an integer file descriptor that refers to an opened file.

pos: specifies a byte offset to be added to a reference location in deriving the new file offset value.

whence: specifies the reference location.

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

****NOTE:**

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

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

1. If a file is opened for read only the lseek will fail.
2. If a file is opened for write access, lseek will succeed and it will extend the file size up to the new file offset address.
3. The return value of lseek is the new file offset address where the next read of write operation will occur, or -1 if lseek call fails.

1.1.8 link:

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

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

cur_link: is a path name of an existing file.

new_link: is a new path name to be assigned to the same file.

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

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

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

1.1.9 unlink:

This function deletes a link of an existing file. It decreases the hard link count attributes of the named file, and removes the file name entry of the link from a directory file. If this function succeeds the file can no longer be referenced by that link. File will be removed by the file system if the hard link count of the file is zero and no process has fdesc referencing that file.

Its prototype is:

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

cur_link: is a path name of an existing file.

- The return value is 0 if it succeeds or -1 if it fails.
- The failure can be due to invalid link name and calling process lacks access permission to remove the path name.

It cannot be used to remove directory files unless the calling process has superuser privilege.ANSI C defines remove function which does the similar operation of unlink. If the argument to the remove functions is empty directory it will remove the directory.

The prototype of rename function is:

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

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

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

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

1.1.10 stat, fstat:

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

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

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

```
struct stat
{
        dev_ts t_dev; //file system ID
        ino_t st_ino; //File inode number
        mode_t st_mode; //contains file type and access flags
        nlink_t st_nlink; //hard link count
```

```
        uid_t st_uid; //file user ID
        gid_t st_gid; //file group ID
        dev_t st_rdev; //contains major and minor device numbers
        off_t st_size; //file size in number of bytes
        time_t st_atime; //last access time
        time_t st_mtime; //last modification time
        time_t st_ctime; //last status change time
    };
```

- The return value of both functions is 0 if it succeeds or -1 if it fails.
- Possible failures may be that a given file path name of file descriptor is invalid, the calling process lacks permission to access the file, or the function interrupted by a signal.
- If a path name argument specified to stat is a symbolic link file, stat will resolve the link and access the non symbolic link file. Both the functions cannot be used to obtain the attributes of symbolic link file.

To obtain the attributes of symbolic link file lstat function was invented. Its prototype is:

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

1.1.11 access:

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

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

path_name: The pathname of a file.

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

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

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

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

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

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

```
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
int main(int argc, char*argv[])
{
        char buf[256];
        int fdesc,len;
        while(--argc>0) {
        if (access(*++argv,F_OK)) { //a brand new file
        fdesc = open(*argv, O_WRONLY|O_CREAT, 0744);
        write(fdesc, "Hello world\n", 12);
        }
        else {
        fdesc = open(*argv, O_RDONLY);
        while(len = read(fdesc, buf,256))
        write(1, buf, len);
        }
close(fdesc); } }
```

1.1.12 chmod, fchmod:

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

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

The prototype of these functions is given below:

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

The chmod function uses path name of a file as a first argument whereas fchmod uses fdesc as the first argument. The flag argument contains the new access permission and any special flags to be set on the file.

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

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

1.1.13 chown, fchown, lchown:

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

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

The function prototypes of these functions are given below:

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

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

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

1.2 File and Record Locking:

UNIX systems allow multiple processes to read and write the same file concurrently which provides data sharing among processes. It also renders difficulty for any process in determining when data in a file can be overridden by another process. In some of the applications like a database manager, where no other process can write or read a file while a process is accessing a database file. To overcome this drawback, UNIX and POSIX systems support a file locking mechanism.

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

A process can impose a write lock or a read lock on either a portion of a file or an entire file. The difference between write locks and read locks is that when a write lock is set, it prevents other processes from setting any overlapping read or write locks on the locked region of a file. On the other hand, when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region of a file. The intention of a write lock is to prevent other processes from both reading and writing the

locked region while the process that sets the lock is modifying the region. A write lock is also known as an *exclusive lock*. The use of a read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region. Other processes are allowed to lock and read data from the locked regions. Hence, a read lock is also called a *shared lock.*

1.2.1 Mandatory Lock

Mandatory locks are enforced by an operating system kernel. If a mandatory exclusive lock is set on a file, no process can use the *read* or *write* system calls to access data on the locked region. If a mandatory shared lock is set on a region of a file, no process can use the *write* system call to modify the locked region. It is used to synchronize reading and writing of shared files by multiple processes: If a process locks up a file, other processes that attempts to write to the locked regions are blocked until the former process releases its lock.

Mandatory locks may cause problems: If a runaway process sets a mandatory exclusive lock on a file and never unlocks it, no other processes can access the locked region of the file until either the runaway process is killed or the system is rebooted. System V.3 and V.4 support mandatory locks.

1.2.2 Advisory Lock

An advisory lock is not enforced by a kernel at the system call level. This means that even though lock (read or write) may be set on a file, other processes can still use the *read* or *write* APIs to access the file. To make use of advisory locks, processes that manipulate the same file must cooperate such that they follow this procedure for every read or write operation to the file:

a. Try to set a lock at the region to be accessed. If this fails, a process can either wait for the lock request to become successful or go do something else and try to lock the file again later.

b. After a lock is acquired successfully, read or write the locked region release the lock must follow the above file locking procedure to be cooperative. This may be difficult to control when programs are obtained from different sources.

All UNIX and POSIX systems support advisory locks.

UNIX System V and POSIX.I use the *fcntl* API for file locking.

The prototype of the *fcntl* API is:

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

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

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

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

The *struct flock* is declared in the <fcntl.h> as:

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

The possible values of l_type are:

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

The possible values of l_whence and their uses are:

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

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

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
int main (int argc, cnar* argv[]) {
        struct flock fvar;
        int fdesc;
        while (--argc > 0) { /* do the following for each file */
        if ((fdesc=open(*++argv,O_RDWR ))==-1 ) {
        perror("open"); continue;
        }
        fvar.l_type = F_WRLCK;
        fvar.l_whence = SEEK_SET;
        fvar.l_start = 0;
        fvar.l_len = 0;
/* Attempt to set an exclusive (write) lock on the entire file */
while (fcntl(fdesc, F_SETLK,&fvar)==-1) {

/* Set lock falls, find out who has locked the file */
    while (fcntl(fdesc,F_GETLK,&fvar)!=-1 && fvar.l_type != F_UNLCK){
    cout<<*argv<<"locked by"<<fvar.l_pid<<"from"<<fvar.l_start<<"for"<<fvar.l_len
<<"byte for"<<(fvar.l_type == F_WRLCK ? 'w':'r')<<endl;
```

```
            if (!fvar.l_len) break;
            fvar.l_start += fvar.l_len;
            fvar.l_len = 0;
            }/* while there are locks set by other processes */
        } /* while set lock un-successful */

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

1.3 Directory File APIs

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

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

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

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

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

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

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

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

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

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

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

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

The uses of these functions are:

opendir:	Opens a directory file for read-only. Returns a file handle DIR* for future reference of the file.
readdir:	Reads a record from a directory file referenced by *dir_fdesc* and returns that record information.
closedir:	Closes a directory file referenced by *dir_fdesc.*
rewinddir:	Resets the file pointer to the beginning of the directory file referenced by *dir_fdesc.*

telldir: Returns the file pointer of a given dir_fdesc.

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

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

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

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

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#if defined (BSD) && !_POSIX_SOURCE
#include <sys/dir.h>
typedef struct direct Dirent;
#else
#include <dirent.h>
typedef struct dirent Dirent;
#endif
int main (int argc, char* argv[ ])
{
        Dirent* dp;
        DIR* dir_fdesc;
        while (--argc > 0 ) { /* do the following for each file */
        if ( !(dir_fdesc = opendir( *++argv ) )) {
        if (mkdir( *argv, S_ IRWXU|S_IRWXG|S_IRWXO) == -1 )
        perror( "opendir" );
        continue;
}
```

```
        /*scan each directory file twice*/
        for (int i=0;i <2 ; i++) {
                for ( int cnt=0; dp=readdir( dir_fdesc );) {
                        if (i) cout << dp->d_name << endl;
                        if (strcmp( dp->d_name, ".") && strcmp( dp->d_name, ".. ") )
                        cnt++; /*count how many files in directory*/
                        if (!cnt) { rmdir( *argv ); break;) /* empty directory */
                        rewinddir( dir fdesc ); / reset pointer for second round */
                        }
        closedir( dir fdesc );
        }
}
```

1.4 Device File APIs

Device files are used to interface physical devices with application programs. Specifically, when a process reads or writes to a device file, the kernel uses the major and minor device numbers of a file to select a device driver function to carry out the actual data transfer.

Device files may be character-based or block-based. UNIX systems define the ***mknod*** API **to create device** files.

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

4. The major and minor device numbers are extended to fourteen and eighteen bits, respectively.

5. In UNIX, if a calling process has no controlling terminal and it opens a character device file, the kernel will set this device file as the controlling terminal of the process. How-ever, if the O_NOCTTY flag is set in the *open* call, such action will be suppressed.

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

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

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
int main( int argc, char* argv[ ] ) {
if(argc!=4){
cout << "usage: " << argv[0] << " <file> <major no> <minor no>\n";
return 0;
}
int major = atoi( argv[2]), minor = atoi( argv[3] );
(void) mknod( argv[1], S_IFCHR | S_IRWXU | S_IRWXG | S_IRWXO, ( major <<8) | minor );
int rc=1, fd = open(argv[1], O_RDWR | O_NONBLOCK | O_NOCTTY );
char buf[256];
while ( rc && fd != -1 )
if (( rc = read( fd, buf, sizeof( buf )) ) < 0 )
perror( "read" );
else if ( rc) cout << buf << endl;
close(fd);
}
```

1.5 FIFO file API's

FIFO files are sometimes called named pipes.

- Pipes can be used only between related processes when a common ancestor has created the pipe.
- Creating a FIFO is similar to creating a file.
- Indeed the pathname for a FIFO exists in the file system.

The prototype of mkfifo is

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

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

Example

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

- The above statement creates a FIFO file "divya" with read-write-execute permission for user and only read permission for group and others.
- Once we have created a FIFO using mkfifo, we open it using open.
- Indeed, the normal file I/O functions (read, write, unlink etc) all work with FIFOs.

When a process opens a FIFO file for reading, the kernel will block the process until there is another process that opens the same file for writing.

Similarly whenever a process opens a FIFO file write, the kernel will block the process until another process opens the same FIFO for reading.

This provides a means for synchronization in order to undergo inter-process communication.

If a particular process tries to write something to a FIFO file that is full, then that process will be blocked until another process has read data from the FIFO to make space for the process to write.

Similarly, if a process attempts to read data from an empty FIFO, the process will be blocked until another process writes data to the FIFO.

From any of the above condition if the process doesn't want to get blocked then we should specify O_NONBLOCK in the open call to the FIFO file.

If the data is not ready for read/write then open returns –1 instead of process getting blocked.

If a process writes to a FIFO file that has no other process attached to it for read, the kernel will send SIGPIPE signal to the process to notify that it is an illegal operation.

Another method to create FIFO files (not exactly) for inter-process communication is to use the pipe system call.

The prototype of pipe is

```
#include <unistd.h>

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

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

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

```
#include<iostream.h>
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<string.h>
#include<errno.h>
int main(int argc,char* argv[])
{
        if(argc!=2 && argc!=3)
        {
                cout<<"usage:"<<argv[0]<<"<file> [<arg>]";
                return 0;
        }
int fd;
```

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

1.6 Symbolic Link File API's

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

The prototype is

```
#include<unistd.h>
 #include<sys/types.h>
#include<sys/stat.h>
 int symlink(const char *org_link, const char *sym_link);
int readlink(const char* sym_link,char* buf,int size);
int lstat(const char * sym_link, struct stat* statv);
```

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

/* Program to illustrate symlink function */

```
#include<unistd.h>
#include<sys/types.h>
 #include<string.h>
 int main(int argc, char *argv[])
{
        char *buf [256], tname [256];
        if (argc ==4) return symlink(argv[2], argv[3]); /* create a symbolic
link */
        else return link(argv[1], argv[2]); /* creates a hard link */
}
```

UNIX PROCESSES and Process Control

INTRODUCTION

We will see how the main function is called when the program is executed, how command line arguments are passed to the new program, what the typical memory layout looks like, how to allocate additional memory, how the process can use environment variables and different way for the process to terminate. We will look at the longjmp and setjmp functions and their interaction with the stack.

main FUNCTION

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

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

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

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

PROCESS TERMINATION

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

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

Abnormal termination occurs in three ways:

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

Exit Functions

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

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

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

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

- Any of these functions is called without an exit status.
- main does a return without a return value
- main "falls off the end", i.e if the exit status of the process is undefined.

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

atexit Function

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

#include <stdlib.h>

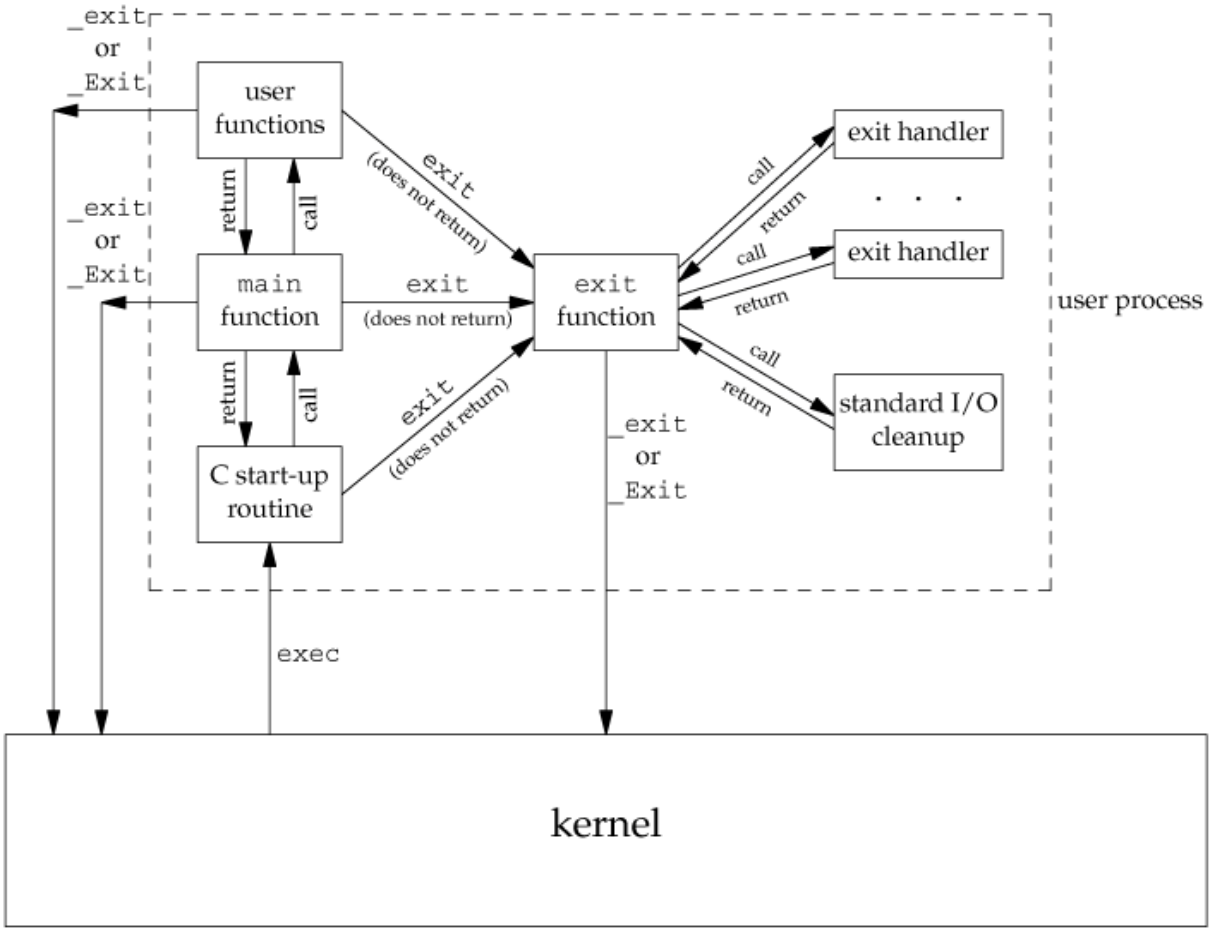
int atexit(void (*func)(void));

Returns: 0 if OK, nonzero on error This declaration says that we pass the address of a function as the argument to atexit. When this function is called, it is not passed any arguments and is not expected to return a value. The exit function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

Example of exit handlers

```
#include<unistd.h>
static void my_exit1(void);
static void my_exit2(void);
 int main(void)
{
        if (atexit(my_exit2) != 0)
                err_sys("can't register my_exit2");
        if (atexit(my_exit1) != 0)
                err_sys("can't register my_exit1");
        if (atexit(my_exit1) != 0)
                err_sys("can't register my_exit1");
        printf("main is done\n"); return(0);
}
static void my_exit1(void)
{
        printf("first exit handler\n");
}
static void my_exit2(void)
{
        printf("second exit handler\n");
}
```

Output: $ ./a.out main is done first exit handler first exit handler second exit handler The below figure summarizes how a C program is started and the various ways it can terminate.



COMMAND-LINE ARGUMENTS

When a program is executed, the process that does the exec can pass command-line arguments to the new program. Example: Echo all command-line arguments to standard output

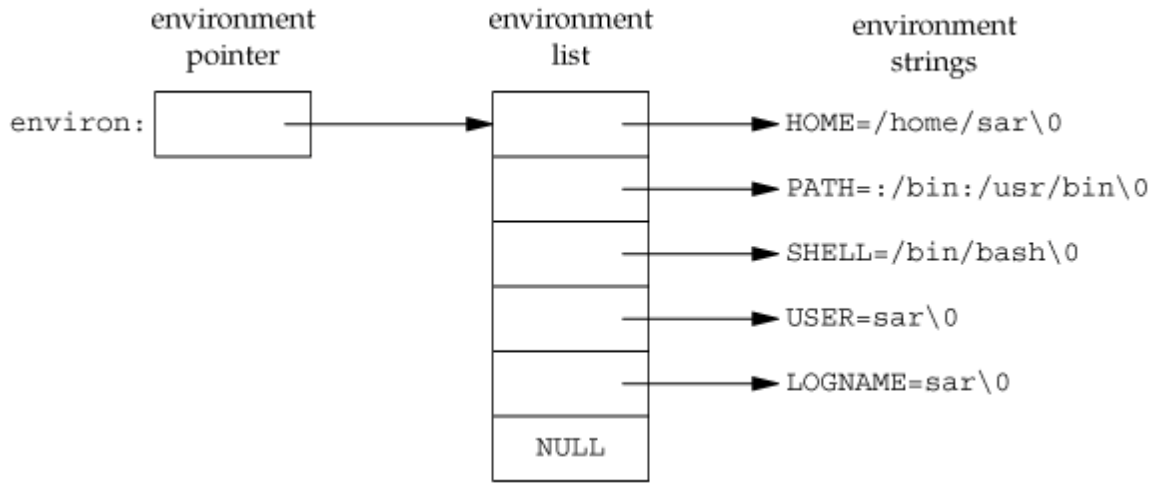
```
#include<unistd.h>
int main(int argc, char *argv[])
{
        int i;
        for (i = 0; i < argc; i++) /* echo all command-line args */
       printf("argv[%d]: %s\n", i, argv[i]);
       exit(0);
 }
```

Output: $ ./echoarg arg1 TEST foo argv[0]:

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

ENVIRONMENT LIST

Each program is also passed an environment list. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string. The address of the array of pointers is contained in the global variable environ: **extern char **environ; Figure : Environment consisting of five C character strings** Generally any environmental variable is of the form: ***name=value.***



MEMORY LAYOUT OF A C PROGRAM

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

- **Text segment**, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.
- **Initialized data segment**, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration

int maxcount = 99;

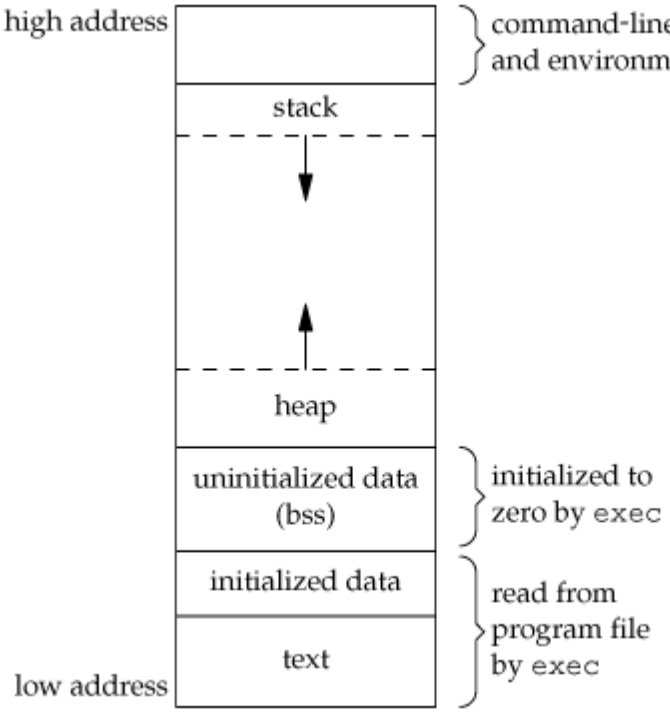
appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

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

  long sum[1000];

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

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



SHARED LIBRARIES

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

MEMORY ALLOCATION

ISO C specifies three functions for memory allocation:

1. malloc: Which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
2. calloc: Which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.

3. realloc: Which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);

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

void free(void *ptr);

The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. Because the three alloc functions return a generic void * pointer, if we #include <stdlib.h> (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type. The function free causes the space pointed to by ptr to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three alloc functions.

The realloc function lets us increase or decrease the size of a previously allocated area. For example, if we allocate room for 512 elements in an array that we fill in at runtime but find that we need room for more than 512 elements, we can call realloc. If there is room beyond the end of the existing region for the requested space, then realloc doesn't have to move anything; it simply allocates the additional area at the end and returns the same pointer that we passed it. But if there isn't room at the end of the existing region, realloc allocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area. The allocation routines are usually implemented with the sbrk(2) system call. Although sbrk can expand or contract the memory of a process, most versions of malloc and free never decrease their memory size. The space that we free is available for a later allocation, but the freed space is not usually returned to the kernel; that space is kept in the malloc pool.

It is important to realize that most implementations allocate a little more space than is requested and use the additional space for record keeping the size of the allocated block, a pointer to the next allocated block, and the like. This means that writing past the end of an allocated area could overwrite this record-keeping information in a later block. These types of errors are often catastrophic, but difficult to find, because the error may not show up until much later. Also, it is possible to overwrite this record keeping by writing before the start of the allocated area. Because memory allocation errors are difficult to track down, some systems provide versions of these functions that do additional error checking every time one of the three alloc functions or free is called. These versions of the functions are often specified by including a special library for the link editor. There are also publicly available sources that you can compile with special flags to enable additional runtime checking.

Alternate Memory Allocators

Many replacements for malloc and free are available.

- **libmalloc**

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

- **vmalloc**

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

- **quick-fit**

Historically, the standard malloc algorithm used either a best-fit or a first-fit memory allocation strategy. Quick-fit is faster than either, but tends to use more memory. Free

implementations of malloc and free based on quick-fit are readily available from several FTP sites.

- **alloca Function**

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

ENVIRONMENT VARIABLES

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

The prototype of these functions are

#include <stdlib.h>

char *getenv(const char *name);

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

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

The prototypes of these functions are

#include <stdlib.h>

int putenv(char *_str_);

int setenv(const char *_name_, const char *_value_, int _rewrite_);

int unsetenv(const char *_name_);

All return: 0 if OK, nonzero on error.

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

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

Environment variables defined in the Single UNIX Specification

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

NOTE:

If we're modifying an existing name:

o If the size of the new value is less than or equal to the size of the existing value, we can just copy the new string over the old string.

o If the size of the new value is larger than the old one, however, we must malloc to obtain room for the new string, copy the new string to this area, and then replace the old pointer in the environment list for name with the pointer to this allocated area.

If we're adding a new name, it's more complicated. First, we have to call malloc to allocate room for the name=value string and copy the string to this area.

o Then, if it's the first time we've added a new name, we have to call malloc to obtain room for a new list of pointers. We copy the old environment list to this new area and store a pointer to the name=value string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally, we set environ to point to this new list of pointers.

o If this isn't the first time we've added new strings to the environment list, then we know that we've already allocated room for the list on the heap, so we just call realloc to allocate room for one more pointer. The pointer to the new name=value string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

setjmp AND longjmp FUNCTIONS

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

#include <setjmp.h>

int setjmp(jmp_buf env);

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

void longjmp(jmp_buf env, int val);

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

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

Example of setjmp and longjmp

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

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

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

getrlimit AND setrlimit FUNCTIONS

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

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

int getrlimit(int resource, struct rlimit *rlptr);

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

Both return: 0 if OK, nonzero on error

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

struct rlimit

{

rlim_t rlim_cur; /* soft limit: current limit */

rlim_t rlim_max; /* hard limit: maximum value for rlim_cur */

};

Three rules govern the changing of the resource limits.

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

An infinite limit is specified by the constant RLIM_INFINITY.

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

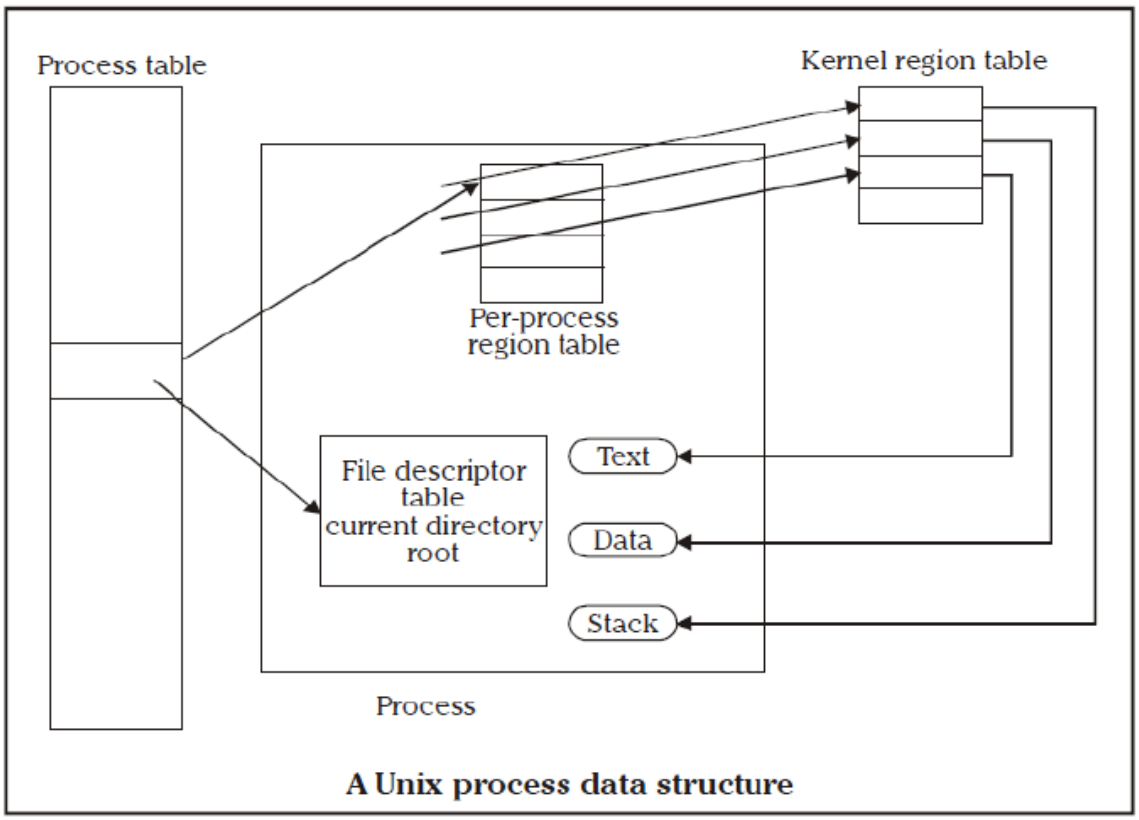
The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes.

UNIX KERNEL SUPPORT FOR PROCESS

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

- A text segment consists of the program text in machine executable instruction code format.
- The data segment contains static and global variables and their corresponding data.
- A stack segment contains runtime variables and the return addresses of all active functions for a process.

UNIX kernel has a process table that keeps track of all active process present in the system. Some of these processes belongs to the kernel and are called as “system process”. Every entry in the process table contains pointers to the text, data and the stack segments and also to U-area of a process. U-area of a process is an extension of the process table entry and contains other process specific data such as the file descriptor table, current root and working directory inode numbers and set of system imposed process limits.



A Unix process data structure

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

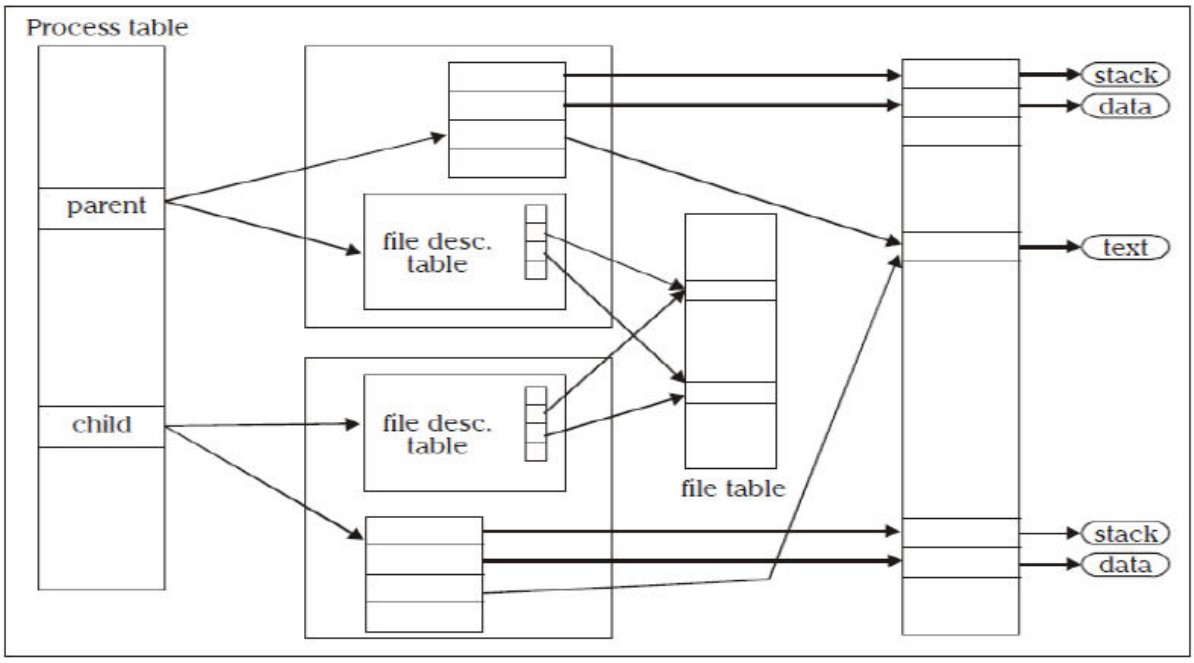


Figure: Parent & child relationship after fork

Figure: Parent & child relationship after fork The process will be assigned with attributes, which are either inherited from its parent or will be set by the kernel.

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

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

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

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

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

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

PROCESS CONTROL

Process identifiers

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

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

Fork function

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

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

- ✓ The new process created by fork is called child process
- ✓ The function is called once but returns twice
- ✓ The return value in the child is 0
- ✓ The return value in parent is the process ID of the new child
- ✓ The child is a copy of parent
- ✓ Child gets a copy of parents text, data , heap and stack
- ✓ Instead of completely copying we can use COW copy on write technique

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

Output

file sharing

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

1. The parent waits for the child to complete

2. After fork the parent closes all descriptors that it doesn't need and the does the same thing

Besides open files the other properties inherited by child are

- ✓ Real user ID, group ID, effective user ID, effective group ID
- ✓ Supplementary group ID

- ✓ Process group ID
- ✓ Session ID
- ✓ Controlling terminal
- ✓ set-user-ID and set-group-ID
- ✓ Current working directory
- ✓ Root directory
- ✓ File mode creation mask
- ✓ Signal mask and dispositions
- ✓ The close-on-exec flag for any open file descriptors
- ✓ Environment
- ✓ Attached shared memory segments
- ✓ Resource limits

The difference between the parent and child

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

The functions of fork

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

Vfork

- ✓ It is same as fork
- ✓ It is intended to create a new process when the purpose of new process is to exec a new program

The child runs in the same address space as parent until it calls either exec or exit

vfork guarantees that the child runs first , until the child calls exec or exit

```
int glob = 6;
int main(void)
{
        int var;
        /* automatic variable on the stack */
        pid_t pid;
        var = 88;
        printf("before vfork\n");
        if ( (pid = vfork()) < 0)
                err_sys("vfork error");
        else if (pid == 0) { /* child */
                glob++;
                /* modify parent's variables */
                var++;
        _exit(0); /* child terminates */

        /* parent */
        printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
        exit(0);
}
```

exit functions

- Normal termination

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

- Abnormal termination

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

- Exit status is used to notify parent how a child terminated
- When a parent terminates before the child, the child is inherited by init process
- If the child terminates before the parent then the information about the is obtained by parent when it executes wait or waitpid
- The information consists of the process ID, the termination status and amount of CPU time taken by process
- A process that has terminated , but whose parents has not yet waited for it, is called a zombie
- When a process inherited by init terminates it doesn't become a zombie
- Init executes one of the wait functions to fetch the termination status

Wait and waitpid functions

- When a child id terminated the parent is notified by the kernel by sending a SIGCHLD signal
- The termination of a child is an asynchronous event
- The parent can ignore or can provide a function that is called when the signal occurs

The process that calls wait or waitpid can

1. Block
2. Return immediately with termination status of the child
3. Return immediately with an error

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

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

Waitpid

The interpretation of pid in waitpid depends on its value

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

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

Waitid

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
Returns: 0 if OK, 1 on error
```

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

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

wait3 and wait4 functions

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

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

Race condition

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

Program with race condition

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

exec functions

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

```
#include <unistd.h>
int exec1 ( const char *pathname, const char *arg0 ,… /*(char *) 0*/);
int execv (const char *pathname, char * const argv[ ]);
int execle (const char *pathname, const char *arg0 ,… /*(char *) 0,
char *const envp[ ] */);
int execve ( const char *pathname, char *const argv[ ] , char *const envp [ ]);
int execlp (const char *filename, const char *arg0 ,… /*(char *) 0*/);
int execvp (const char *filename ,char *const argv[ ] );
```

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

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

REFERENCES

1. W. Richard Stevens: Advanced Programming in the UNIX Environment, 2nd Edition, Pearson Education, 2005 (Chapter 3,7,8,10,13,15)
2. Unix System Programming Using C++ - Terrence Chan, PHI, 1999. (Chapter 7,8,9,10)

Web/ Video Resources

1. General file API's (create,open,read,write and close)

   https://youtu.be/pX_6d1z972k
2. Demonstration on General file API's (open, read, write and close)

   https://youtu.be/esROGmkMIhg
3. General file API's part-2

   https://youtu.be/S-wYT5fn1K0
4. Demonstration on General file API's part-2

   https://youtu.be/PdI2G6E4DM0
5. File and Record locking Directory, Device ,FIFO ,Symbolic link file API's

   https://youtu.be/I1GLAUtupSc
6. The Environment of a UNIX Process

   https://youtu.be/4bfzEyb4YD0
7. Process Control

   https://youtu.be/bfOUZN5mh5w

Question Bank

UNIX File APIs

1. Explain some general file API's with their prototypes and argument values.
2. Explain file and record locking with an example program. (LAB PGM)
3. Write a program to implement ln and mv command in UNIX.
4. What is FIFO file? Write a program to implement FIFO file.
5. Implement symbolic link file API.

UNIX PROCESSES

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

2. With an example explain the use of setjmp and longjmp functions.

3. Describe the UNIX Kernel support for process. Show the related data structures.

4. Bring out the importance of locking files. What is the drawback of advisory lock? Explain in brief.

5. 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.

6. With a neat diagram, explain the memory layout of c program. In which segments are the automatic variables and dynamically created objects are stored?

7. What are setjmp and longjmp function? Explain with a program to transfer the control across functions using them.

PROCESS CONTROL

1. Explain the following system calls: i) fork ii) vfork iii) exit iv) wait.

2. Giving the prototype explain different variant of exec system call.

3. What is race condition? Write a program in C/C++ to illustrate a race condition