UNIX PROGRAMMING :Module3-NOTES

Module – 3- Syllabus

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

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

Process Control : Introduction, Process Identifiers, fork, vfork, exit, wait, waitpid, wait3, wait4 Functions, Race Conditions, exec Functions.

Module- 3

UNIX FILE API'S

3.1 General File APIs

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

FILE APIs	USE
open ( )	This API is used by a process to open a file f r data access.
read ( )	The API is used by a process to read data fr m a file
write ( )	The API is used by a process to write data t a file
lseek ( )	The API is used by a process to allow rand m access to a file
close ( )	The API is used by a proce to terminate onnection to a file
stat ( ) fstat ( )	The API is used by a proce to query file attributes
chmod ( )	The API is used by a pro e to hange file access permissions.
chown ( )	The API is used by a pro e to hange UID and/or GID of a file
utime ( )	The API is used by a pro e to hange the last modification and access time stamps of a file
link ( )	The API is used by a pro ess to reate a hard link to a file.
unlink ( )	The API is used by a process to delete hard link of a file
umask ( )	The API is used by a process to set default file creation mask.

Open:

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

Prototype:

#include < sys/types.h>

#include <unistd.h>

#include <fcntl.h>

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

path name : The pathname of a file to be opened or created. It can be an absolute path name or

relative path name. The pathname can also be a symbolic link name.

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

Access mode flags:

O_RDONLY: Open the file for read only. If the file is to be opened for read only then the file should already exist in the file system and no modifier flags can be used.

O_WRONLY: Open the file for write only. If the file is to be opened for write only, then any of the access modifier flags can be specified.

x O_RDWR: Open the file for read and write. If the file is to be pened f r write only, then any of the access modifier flags can be specified.

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

Access Modifier Flags:

x O_APPEND: Appends data to the end of the file

x O_CREAT : Create the file if it does not exist. If the file exists it has no effects.
However

if the file does not exist and O CREATE is not spe ified, open will abort with a failure return status.

O_EXCL : Used with O CREAT, if he file exists, the call fails. The test for existence and the creation if the file does not exis s.

O_TRUNC: If the file exits, discards he file contents and sets the file size to zero.

- x O NOCTTY : Species not to use the named terminal device file as the calling process control terminal.
- O NONBLOCK: Specifies that any subsequent read or write on the file should be non-blocking. Example, a process is normally blocked on reading an empty pipe or on writing to a pipe that is full. It may be used to specify that such read and write operations are non-blocking.

Example:

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

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

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

Permission:

The permission argument is required only if the O_CREAT flag is set in the access_mode argument. It specifies the access permission of the file for its owner, group and all the other people.

Its data type is int and its value is octal integer value, such as 0764. The left-most, middle and right-most bits specify the access permission for owner, group and thers 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 f r ther.

7= 111 specifies read, write and execution permi ion 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:

IRWXU|S IRGRP|S IWGRP|S_IROTH

Permission value is modified by i s 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 ne mask value as argument, which is used by calling process and the function returns the old umask value. For example,

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

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

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

actual_permission = permission & ~umask_value

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

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

Creat:

The creat system call is used to create new regular files. Its pr t type is:

#include < sys/types.h>

#include <unistd.h>

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

The path_name argument is the path name of a file to be reated

The mode argument is same as that for open API.

Since O_CREAT flag was added to open API it wa u ed to both create and open regular files. So, the creat API has become obsol te. It is retained for backward-compatibility with early

versions of UNIX.

The creat function can be implemen ed sing he open function as:

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

read:

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

Its prototype is:

#include <sys/types.h>

#include <unistd.h>

ssize t read (int fdesc ,void* buf, size_t size);

–

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

buf: is the address of a buffer holding any data read.

size: specifies how many bytes of data are to be read from the file.

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

pointer (void *). For example the following code reads, sequentially one or more record of struct

sample-typed data from a file called dbase:

struct sample { int x; double y; char* a;} varX;

int fd = open("dbase", O_RDONLY);

while (read(fd, &varX, sizeof(varX))>0)

The return value of read is the number of bytes of data successfully read and stored in the buf argument. It should be equal to the size value.

If a file contains less than size bytes of data remaining to be read, the return value of read will be less than that of size. If end-of-file is reached, read will return a zero value.

□□size_t□is defined as□int□in <sys/types.h> header, u er hould 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:

The read function will return -1 val e, errno will be set to EINTR, and all the data will be discarded.

The read function will return he number of bytes of data read prior to the signal interruption. This allows a process to continue reading the file.

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

write:

The rite 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 su essfully written to a file. It should be equal to the *size* value.

□□ □ □ □ □ □ □ □

f the write will cause the file size to exceed a y tem imposed limit or if the file system disk is full, the return value of write will be the a tual number of bytes written before the function was aborted.

□□ □ □ □ □ □ □ □

f a signal arrives during a write f nction all and the OS does not restart the system call automatically, the write function may ei her ret rn a -1 value and set errno to EINTR or return the number of bytes of data writ en prior o he signal interruption.

□□ □ □ □ □ □ □ □

he write function may perform nonblocking operation if the O_NONBLOCK or O_NDELAY flags are set on the fdesc argument to the function.

close:

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

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

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

The return value of close is zero if the call succeeds or -1 if it fails.

The close function frees unused file descriptors so that they can be reused to reference other files.

The close function will deallocate system resources which reduces the memory requirement of a process.

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

fcntl:

The fcntl function helps to query or set access control flags and the close-on-exec flag of any file descriptor. Users can also use fcntl to assign multiple file descript rs t 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 spe ified after cmd is dependent on the actual cmd value.

The possible cmd values are defined in he <f ntl.h> header. These values and their uses are:

cmd value	Use
F_GETFL	Returns he 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.
_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.
_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.
_DUPFD	Duplicates the file descriptor fdesc with another file descriptor. The third argument to fcntl is an integer value which specifies that the duplicated file descriptor must be greater than or equal to that value. The return value of fcntl, in this case is the duplicated file descriptor.

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

int cur_flags = fcntl(fdesc, FGETFL);

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

The close-on-exec flag of a file descriptor specifies that if the process that owns the descriptor

calls the exec API to execute different program, the fdesc sh uld be cl sed by the kernel before the new program runs or not.

The example reports the close-on-exec flag of a fdesc, sets it t n afterwards:

```
cout<<fdesc<<"close-on-exec:"<<fcntl(fdesc,F_GETFD)<<endl;
(void)fcntl(fdesc, F_SETFD, 1);
```

The fcntl function can also be used to dupli ate a fde c with another fdesc. The results are two fdesc reference the same file with same a es mode and are the same file pointer to read or write the file. This is useful in the redirection of tandard input or output to reference a file.

Example: Reference standard input of a pro ess to a file called FOO

```
int fdesc = open("FOO", O_RDONLY);     //open FOO for read
close(0);                              //close standard input
if(fcntl(fdesc, F_DUPFD, 0)==-1) perror("fcntl");          //stdin from FOO
char buf[256];
int rc = read(0,buf,256);                                  //read data from FOO
```

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

```
#define dup(fdesc)              fcntl(fdesc, F_FUPFD,0)
 #define dup2(fdesc1,fd2)       close(fd2),fcntl(fdesc, F_DUPFD, fd2)
```

The dup function duplicates a fdesc with the lowest unused fdesc of a calling process.
The dup2 function will duplicate a fdesc using a fd2 fdesc, regardless of whether fd2 is used to reference another file.

lseek:

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

Its prototype is:
#include <sys/types.h>
#include <unistd.h>
off_t lseek (**int** fdesc , **off_t** pos, **int** whence);
fdesc: is an integer file descriptor that refers to an opened file.
pos: specifies a byte offset to be added to a reference lo ati n in deriving the new file offset value.
whence: specifies the reference location.

Whence value	Reference location
SEEK_CUR	urrent 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 nega i e pos value with the whence value set to SEEK_SET as this will set negative offset.

b. If an lseek call ill 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 rite access, lseek will succeed and it will extend the file size up to the new file offset address.

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

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

```
#include<iostream.h>
#include<sys/types.h>
#include<unistd.h>
streampos iostream::tellg()
{
        return (streampos)lseek(this->fileno(),(off_t)0,SEEK_CUR);
}
iostream&iostream::seekg(streampos pos,seek_dir ref_loc)
{
if(ref_loc == ios::beg)
(void)lseek(this->fileno(), (off_t)pos, SEEK_SET);
else if(ref_loc == ios::cur)
        (void)lseek(this->fileno(), (off_t)pos, SEEK_CUR);
else if(ref_loc == ios::end)
        (void)lseek(this->fileno(), (off_t)po , SEEK_END);
return *this;
}
```

The iostream::tellg simply calls lseek to ret rn the current file pointer associated with an iostream object. The file descriptor of an iostream object **const char*** is obtained from the fileno member function.

The iostream::seekg relies on lseek o al er he file pointer associated with an iostream object. The arguments are file offset and a reference location for the offset. This function also converts seek_dir value to an lseek hence alue.

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

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

link:

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

#include <unistd.h>

int link (**const char*** cur_link ,**const char*** new_link)

cur_link: is a path name of an existing file.

---

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

If this call succeeds, the hard link count attribute of the file will be increased by 1.

link cannot be used to create hard links across file systems. It cannot be used on directory files unless it is called by a process that has superuser previlege.

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

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

unlink:

This function deletes a link of an existing file. It decreases the hard link count attributes of the named file, and removes the file name entry of the link from a directory file. If this function succeeds the file can no longer be referenced by that link.

File ill 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 na e)

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(arg [1],argv[2])) cerr<<"usage:"<<arg
            [0]<<"<old link><new_link>\n"; _
        else if(link (arg [1], arg [2])==0)
            return unlink(argv[1]);
    return -1;
}
```

stat, fstat:

These functions retrieve the file attributes of a given file. The first argument of *stat* is file path name here 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 a  e      time
    time_t      st_mtime;            //last modifi ation time
    time_t      st_ctime;            //last stat s    hange time
  };
```

The return value of both functions is 0 if i succeeds or -1 if it fails.

Possible failures may be that a gi en file path name of file descriptor is invalid, the calling process lacks permission to access the file, or the function interrupted by a signal.

*x*If a path name argument specified to stat is a symbolic link file, stat will resolve the link and access the non symbolic link file. Both the functions cannot be used to obtain the attributes of symbolic link file.

To obtain the attributes of symbolic link file lstat function was invented. Its prototype is: int lstat (const char* path_name,struct stat* statv)

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

```
#include <iostream.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
static char xtbl[10] = "rwxrwxrwx";

static void display_file_type ( ostream& ofs, int st_mode )
{
    switch (st_mode &S_IFMT)
   {
        case S_IFDIR:    ofs << 'd'; return;          /* direct ry file */
        case S_IFCHR:    ofs << 'c'; return;          /* character device file */
        case S_IFBLK:    ofs << 'b'; return;          /* blo k devi e file */
        case S_IFREG:    ofs << ' '; return;          /* regular file */
        case S_IFLNK:    ofs << 'l'; return;          /* symboli  link file */
        case S_IFIFO:    ofs << 'p'; return;          /* FIFO file */
    }
```

```
}
/* Show access permission for owner, gro p, others, and any special flags */
static void display_access_perm ( os ream& ofs, int st_mode ) {

        char amode[10];
        for (int i=0, j= (1 << 8); i < 9; i++, j>>=1)
        amode[i] = (st_mode&j) ? xtbl[i] : '-';          /* set access permission */
        if (st_mode&S_ISUID) amode[2] = (amode[2]=='x') ? 'S' : 's';
        if (st_mode&S_ISGID) amode[5] = (amode[5]=='x') ? 'G' : 'g';
        if (st_mode&S_ISVTX) amode[8] = (amode[8]=='x') ? 'T' : 't';
        ofs << amode << ' ';
}
/* List attributes of one file */
static void long_list (ostream& ofs, char* path_name)
{
    struct stat          statv;
    struct group*gr_p;
    struct passwd*pw_p;
    if (lstat (path_name, &statv))
    {
        cerr<<"Invalid path name:"<< path_name<<endl;
        return;
     }
    display_file_type( ofs, statv.st_mode );
    display_access_perm( ofs, statv.st_mode );
    ofs << statv.st_nlink;                    /* display hard link c  unt */
    gr_p = getgrgid(statv.st_gid);            /* convert GID to gr  up na  e */
    pw_p = getpwuid(statv.st_uid);           /*convert UID to user name */

    ofs << ' ' <<(pw_p->pw name ? pw p->pw name: tatv st uid)
        << ' ' <<(gr_p->gr name ? gr p->gr name: tatv      t gid)<< ' ';
    if ((statv.st_mode&S_IFMT) == S IFCHR || ( tatv    t mode&S_IFMT)==S_IFBLK)
        ofs << MAJOR(statv.st rdev) << ',' <<  MINOR( tatv.st_rdev);
   else ofs << statv.st_size;                          /* show file size or major/minor no. */
    ofs << ' ' << ctime (&statv.st m ime);             /* print last modification time */
```

```
    ofs << ' ' << path_name << endl;                    /* show file name */
}
/* Main loop to display file attributes one file at a time */
int main (int argc, char* arg []) {

        if (argc==1)
            cerr << "usage: " << argv[0] << " <file path name> ...\n";
        else hile (--argc >= 1) long_list( cout, *++argv);
            return 0;
}
```

access:

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

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

path_name: The pathname of a file.

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

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

The flag argument value to access call is composed by bitwise-ORing ne 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 f nction ret rn 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 re urns 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;
      hile(--argc>0) {
       if (access(*++argv,F_OK)) {                    //a brand new file
          fdesc = open(*argv, O_WRONLY|O_CREAT, 0744);
          write(fdesc, "Hello world\n", 12);
       }
        else  {
        fdesc = open(*argv, O_RDONLY);
        while(len = read(fdesc, buf,256))
           write(1, buf, len);
                }
                close(fdesc);
    }
}
```

chmod, fchmod:

The chmod and fcmod functions change file a e permi ions for owner, group and others and also set-UID, set-GID and sticky flags.

A process that calls one of these functions sho ld have the effective user ID of either the super

user or the owner of the file.

The prototype of these functions is gi en below:

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

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

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

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

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

```
#include <unistd.h>

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

chown, fchown, lchown:

The chown and fchown functions change he ser ID and group ID of files. They differ only in their first argument which refer to a file by ei her 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 cho  n (const char* path_name, uid_t uid, gid_t gid);
int fcho  n (int fdesc, uid_t uid, gid_t gid);
int lcho  n (const char* path_name, uid_t uid, gid_t gid);
```

path_name: is the path name of a file.

uid: specifies the new user ID to be assigned to the file.

gid : specifies the new group ID to be assigned to the file.

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

3.2 File and Record Locking:

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

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

File locking is applicable only for regular files. It allows a pr ess t i pose a lock on a file so that other processes cannot modify the file until it is unl ked by the process.

A process can impose a write lock or a read lock on either a p rti n f a file or an entire file. The difference between write locks and read lock i 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 prevent 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 ha se s he lock is modifying the region. A write lock is also known as an *exclusive lock*.

The use of a read lock is to pre en o her processes from writing to the locked region while the process that sets the lock is reading data from the region. Other processes are allowed to lock and read data from the locked regions. Hence, a read lock is also called a *shared lock*.

3.2.1 Mandatory Lock

Mandatory locks are enforced by an operating system kernel.

If a mandatory exclusive lock is set on a file, no process can use the *read* or *write* system calls to access data on the locked region.

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

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

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

3.2.2 Advisory Lock

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

This means that even though lock (read or write) may be set n a file, ther 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 a e ed If this fails, a process can either wait for the lock request to become s ccessf l or go do omething else and try to lock the file again later.

   After a lock is acquired successf lly, read or write the locked region release the lock

   The drawback of ad isory locks are that programs that create processes to share files must follow the abo e file locking procedure to be cooperative. This may be difficult to control when programs are ob ained 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 hich operation is to be performed.

cmd Flag

Use

_
F SETLK F_SETLKW F_GETLK

Sets a file lock. Do not block if this cannot succeed immediately

Sets a file lock and blocks the calling process until the lock is acquired

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 referen e address
        off_t l_len;      // how many bytes in the locked region
        pid_t l_pid;      //PID of a process which ha locked the file
};
```

The possible values of l_type are:

l_ type **value**	**Use**
F_RDLCK	Sets a a read (shared) lo k on a specified region
F_WRLCK	Sets a wri e (excl sive) lock on a specified region
F_UNLCK	Unlocks a specified region

The possible values of l_ hence 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

3.2.3 Lock Promotion and Lock splitting:

If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512.

The previous read lock from 0 to 256 is now covered by the write lock, and the process does not own two locks on the region from 0 to 256. This process is called lock promotion.

Furthermore, if the process now unlocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called lock splitting.

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

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

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


int main (int argc, cnar* argv[]) {
struct flock            fvar;
int                     fdesc;
while (--argc > 0) {                    /* do the following for ea h file */
 if ((fdesc=open(*++argv,O RDWR ))==-1 ) {
perror("open"); continue;
}
fvar.l_type       = F_WRLCK;
fvar.l_whence = SEEK_SET;
fvar.l_start      = 0;
fvar.l_len        = 0;
/* Attempt to set an exclusive (write) lock on the entire file */
while (fcntl(fdesc, FSETLK,&fvar)==-1) {
/* Set lock falls, find out    ho has locked the file */
while (fcntl(fdesc,F GETLK,&fvar)!=-1 && fvar.l_type != F_UNLCK){
cout<<*argv<<"locked by"<<fvar.l_pid<<"from"<<fvar.l_start<<"for"<<fvar.l_len
      <<"byte for"<<(fvar.l type == F_WRLCK ? 'w':'r')<<endl;
if (!fvar.l len) break;
```

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

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

3.3 Directory File APIs

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

They are also used by the operating system to onvert 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 );
```

The path name argument is the path name of a directory to be created.

The mode argument specifies the access permission for the owner, group and others to be assigned to the file.

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. Ret rns a file handle DIR* for future reference of the file.

readdir: Reads a record from a direc ory file referenced by *dir_fdesc* and returns that record information.

closedir: Closes a directory file referenced by *dir fdesc.*

rewinddir: Resets the file pointer to the beginning of the directory file referenced by *dir_fdesc.* The next call to *readdir* ill read the first record from the file.

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

telldir: Returns the file pointer of a given dir_fdesc.

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

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

#include <unistd.h>

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

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

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

int main (int argc, char* argv[ ])
 {
Dirent* dp;
DIR*  dir_fdesc;
while (--argc > 0 ) {          /* do the following for each file */
if ( !(dir fdesc = opendir( *++argv ) )) {
if (mkdir( *argv, S  IRWXU|S IRWXG|S_IRWXO) == -1 )
 perror( "opendir" );
continue;
}
/*scan each directory file twice*/
for (int i=0;i < 2 ; i + + ) {
for ( int cnt=0; dp=readdir( dir_fdesc );) {
if (i) cout << dp->d name << endl;
if (strcmp( dp->d name, ".") && strcmp( dp->d_name, ".. ") )
 cnt++;                              /*count how many files in directory*/
```

```
if (!cnt) { rmdir( *argv ); break;) /* empty directory */
   rewinddir( dir fdesc );   / reset pointer for second round */
   }
   closedir( dir fdesc );
  }
}
```

3.4 Device File APIs

Device files are used to interface physical devices with applicati n pr gra s.

Specifically, when a process reads or writes to a devi e file, the kernel uses the major and minor device numbers of a file to select a device driver fun ti n t carry out the actual data transfer.

Device files may be character-based or block-ba ed

UNIX systems define the mknod API to reate devi e file .

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

The path_name argumen is he pa h name of a directory to be created.

The mode argument specifies he access permission for the owner, group and others to be assigned to the file

The *device id* contains the major and minor device numbers and is constructed in most UNIX systems as follows: The lowest byte of *a device_id* is set to a minor device number and the next byte is set to the major device number. For example, to create a block device file called SCSI5 ith major and minor numbers of 15 and 3, respectively, and access rights of read-write-execute for everyone, the *mknod* system call is:

mknod("SCSI5", S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO, (15<<8) 13);

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

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.

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 = a oi( 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);
}
```

```
        pid_t  pid;
        FILE    *fp;
        /* only allow "r" or "w" */
        if ((type[0] != 'r' && type[0] != 'w') || ype[1] != 0) { errno
           = EINVAL; /* req ired by POSIX */ return(NULL);
        }

        if (childpid == NULL) { /* first time through */ /*
           allocate zeroed out array for child pids */ maxfd =
           sysconf( SC OPEN_MAX);
           if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
              return(NULL);
        }
```

```
    if (pipe(pfd) < 0)
        return(NULL);  /* errno set by pipe() */

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

            close(pfd[1]);

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

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


int pclose(FILE *fp)
{

    int
    pid_
fd, stat;
t  pid;
```

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

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

childpid[fd] = 0;


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

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

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

UNIX PROCESSES

Introduction

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

main function

A C program starts execution with a function called main. The prototype for the main function is **int main(int argc, char *argv[]);**

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

When a C program is executed by the kernel by one of the exec functions, a special start-up routine is called before the main function is called.

The executable program file specifies this routine as the starting address for the program, this is set up by the link editor when it is invoked by the C compiler.

This start-up routine takes values from the kernel, the command-line arguments and the environment list and sets things up so that the main function is called.

Process Termination

There are five ways for a process to terminate.

Normal termination occurs in 3 ways:
Abnormal termination occurs in 2 ways:

Normal termination

Return from main
Calling exit
Calling _exit or _Exit

Abnormal termination

Calling abort
Terminated by signal

Exit Functions

Three functions terminate a program normally:

_exit and _Exit, which return to the kernel immediately.

exit, which performs certain cleanup processing and then returns to the kernel.

Function prototype is

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

Prototype of 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.

Example of exit handlers

```
#include "apue.h"
static void my_exit1(void);
static void my_exit2(void);

int main(void)
{
        if (atexit(my_exit1) != 0)
        err_sys ("can't register my_exit1");
```

```
        if (atexit(my_exit2) != 0)
        err_sys ("can't register my_exit2");

        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
      second exit handler
```

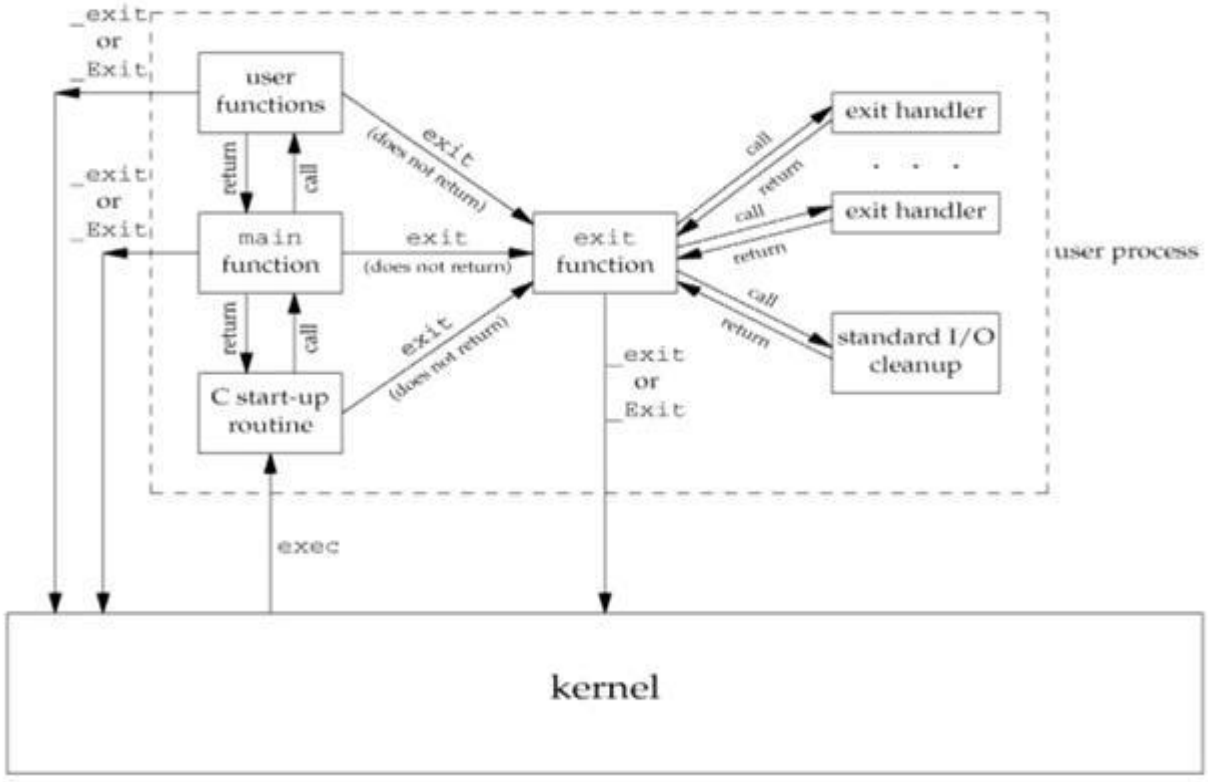


Fig: view of c program starting and the various way to terminate

Command-Line Arguments

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

Example: Echo all command-line arguments to standard output

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

Output:

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

Environment List

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

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

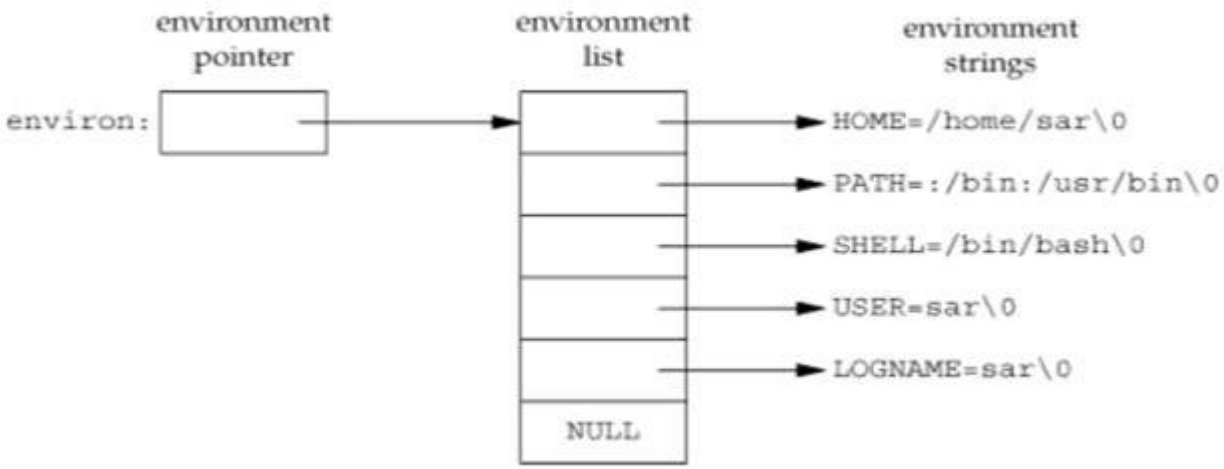
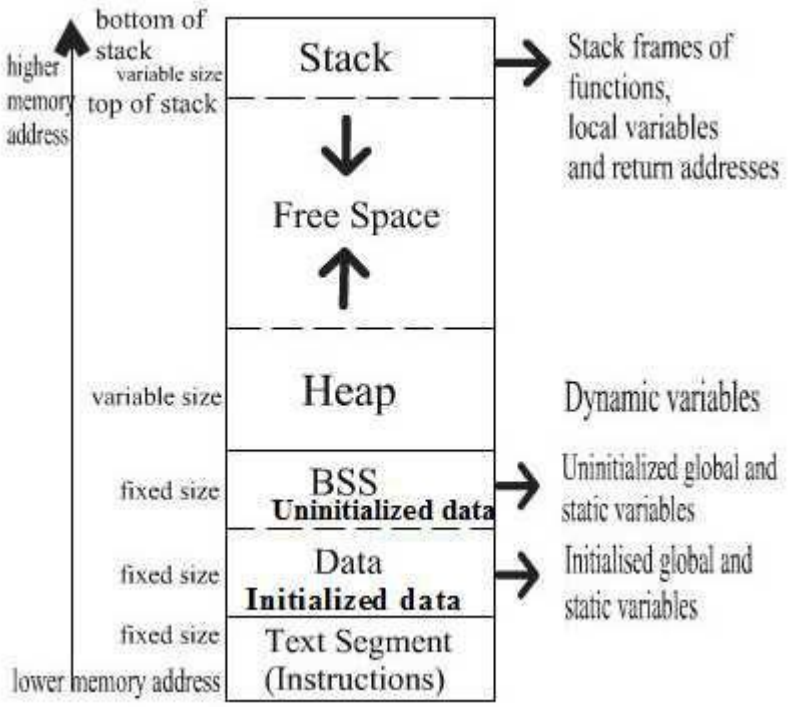


Fig: Environment consisting of five C character strings

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

Memory Layout of a C Program



A C program has been composed of the following pieces:

Text segment – consists of 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 – also simply called the data segment, containing variables that are specifically initialized in the program.

For example, the C declaration **int maxcount = 99;**
This variable to be stored in the initialized data segment with its initial value.

Uninitialized data segment - also 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.

For example, the C declaration **long sum [1000]**
This variable to be stored in the uninitialized data segment.

Heap - where dynamic memory allocation usually takes place.
The heap has been located between the uninitialized data and the stack.

Stack - where automatic variables are stored, along with information that is saved each time a function is called.
When each time a function is called, the address of where to return 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.

Shared Libraries

Nowadays most UNIX systems support shared libraries.
Shared libraries remove the common library routines from the executable file, instead it maintains a single copy of the library routine somewhere in memory that all processes can refer. 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

There are three functions specified by ANSI C 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 the initial value of the space between the old contents and the end of the new area is indeterminate.

Prototype of malloc, calloc, realloc function

```
#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 pointer, if we include #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** deallocated the space pointed to by **ptr.**
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 increase or decrease the size of a previously allocated area.

For example, if we allocate room for 512 elements in an array, that will fill 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** system call. This system call expand the memory of a process.
Although sbrk can expand 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 –which includes size of the allocated block, a pointer to the next allocated block.
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 (dangerous), and 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.

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 shell uses number of environment variables. 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 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.
getenv is used 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

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.

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
HOME	home directory
LANG	name of locale language
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
LOGNAME	login name
PATH	list of path prefixes to search for executable file
PWD	absolute pathname of current working directory
SHELL	name of user's preferred shell
TERM	terminal type
TMPDIR	pathname of directory for creating temporary files
TZ	time zone information

Note:

if we're modifying an existing name:

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

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

If we're adding a new name, it's more complicated.

First, we have to call malloc to allocate room for the name=value string and copy the string to this area.

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

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

setjmp and longjmp functions

In C, we can't goto a label that's in another function. Instead, we must use the setjmp and longjmp functions to perform this type of branching.

These two functions are useful for handling error conditions that occur in a deeply nested function call.

The prototypes of these functions are

```
#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 records the necessary information needed to continue execution.
The env is jmp_buf type defined in <setjmp.h>.

Example of setjmp and longjmp

```
#include "apue.h"
#include<setjmp.h>
#define TOK_ADD 5

Jum_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.

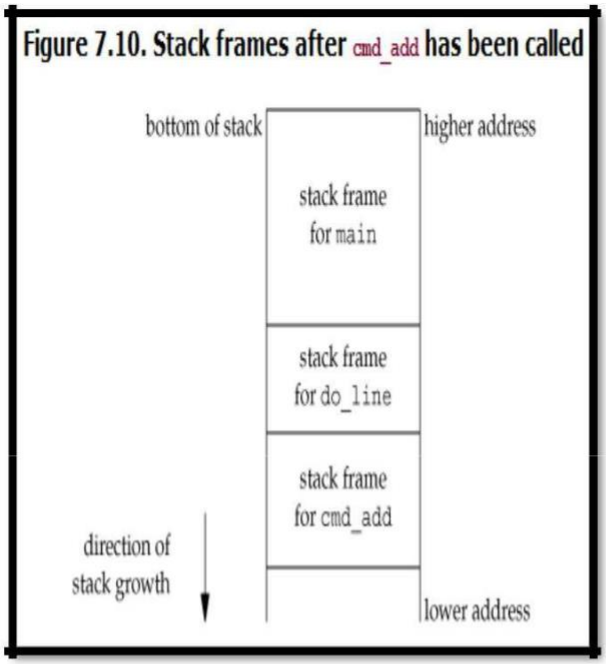
The program code marked by the env must be in a function that is among the callers of the current function.

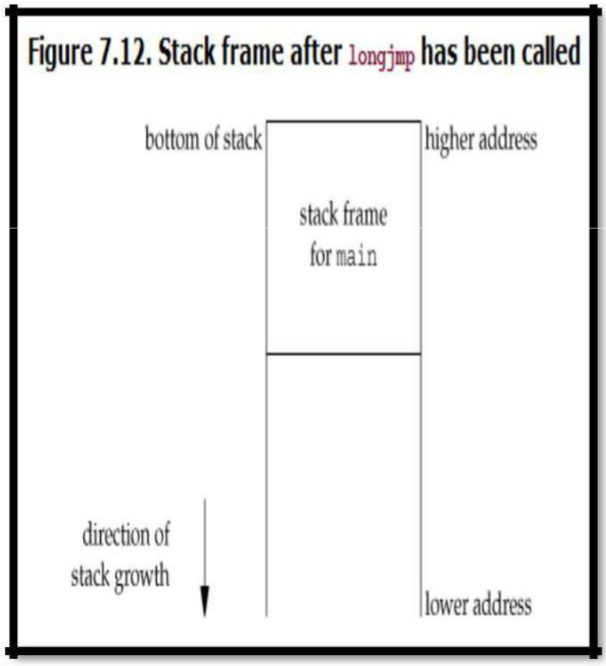
When the process is jumping to the target function, all the stack space used in the current function and its callers, upto the target function are discarded by the longjmp function.

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.

The prototypes of these functions are

```
#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 for changing 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.

The source argument takes one of the following values in above prototype.

RLIMIT_AS	The maximum size in bytes of a process total available memory.
RLIMIT_CORE	The maximum size in bytes of a core file.
RLIMIT_CPU	The maximum amount of CPU time in seconds.
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.
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.
RLIMIT_NOFILE	The maximum number of open files per process.
RLIMIT_NPROC	The maximum number of child processes per real user ID.

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	The maximum size in bytes of the mapped address space.

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

Example: program to print the current soft limit and hard limit.

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

#define doit(name)   pr_limits(#name, name)
static void pr_limits(char *, int);

int main(void)
{
doit(RLIMIT_CORE);
doit(RLIMIT_CPU);
doit(RLIMIT_DATA);
doit(RLIMIT_FSIZE);
```

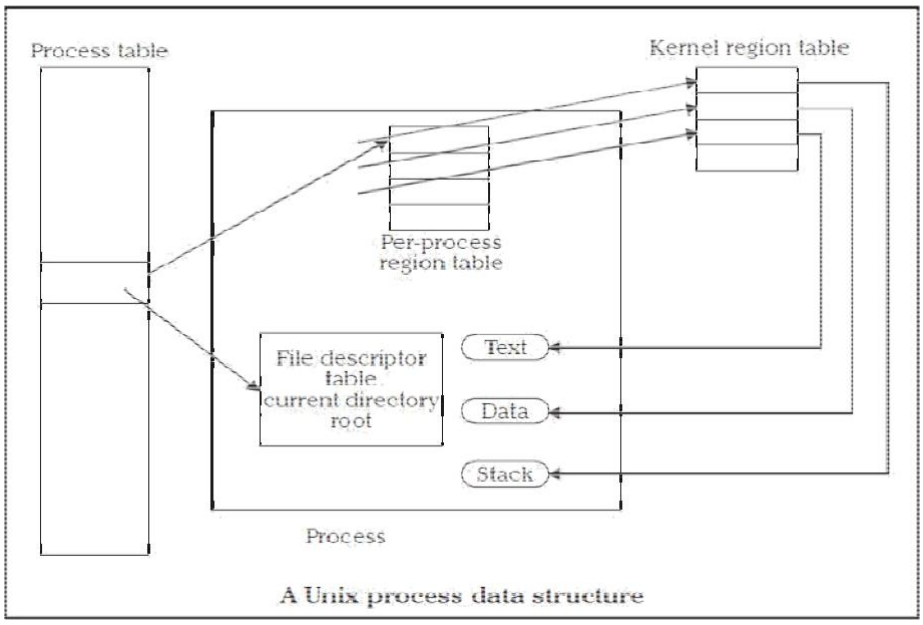
```
#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
#ifdef  RLIMIT_NOFILE    /* SVR4 name */
    doit(RLIMIT_NOFILE);
#endif
#ifdef  RLIMIT_OFILE     /* 4.3+BSD name */
    doit(RLIMIT_OFILE);
#endif
#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
    doit(RLIMIT_STACK);
#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}
```

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

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite)  ");
    else
        printf("%10ld  ", limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)\n");
    else
        printf("%10ld\n", limit.rlim_max);
}
```

UNIX kernel support for process

The data structure and execution of processes are dependent on operating system implementation.



A Unix process data structure

A UNIX process consists 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 stack and a stack provides storage for function arguments, automatic variables, and 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". But majority of processes are associates with the users who are logged in are called user 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 etc.

All processes in UNIX system except the process that is created by the system boot code, are created by the fork system call.

After the fork system call, once the child process is created, both the parent and child processes resumes execution.

When a process is created by fork, it contains duplicated copies of the text, data and stack segments of its parent as shown in the Figure below. Also it has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

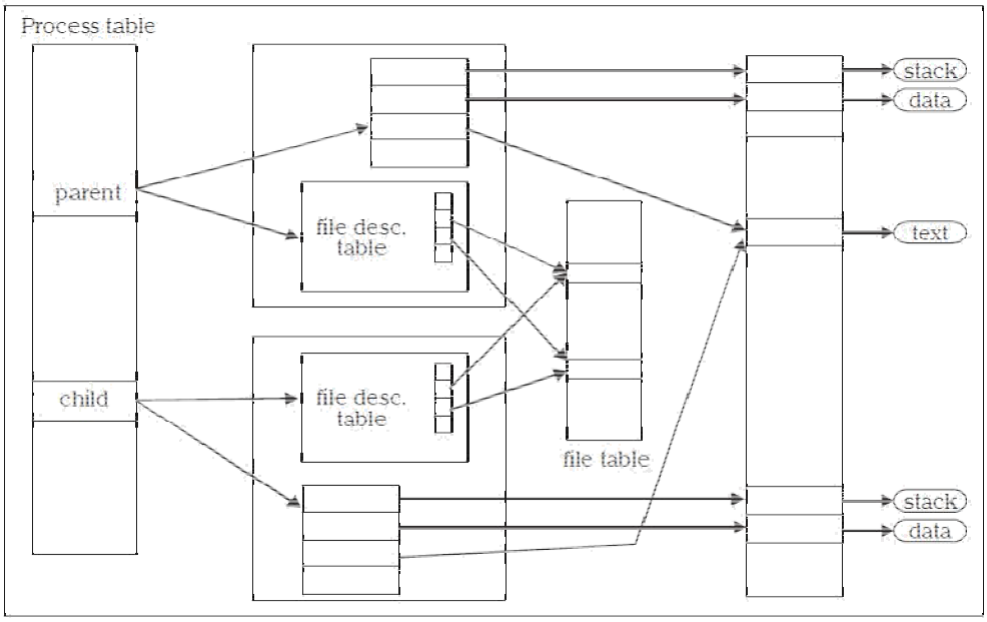


Figure: Parent & child relationship after fork

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

A real user identification number (rUID): the user ID of a user who created the parent process. This is used by the kernel to keep track of who creates which processes on the system.

A real group identification number (rGID): the group ID of a user who created that parent process. This is used by the kernel to keep track of which group creates which processes on the system.

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. This is reset to none in the child 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.

Process control

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

Process identifiers

#include <unistd.h>

pid_t getpid(void);

Returns: process ID of calling process

pid_t getppid(void);

Returns: parent process ID of calling process

uid_t getuid(void);

Returns: real user ID of calling process

uid_t geteuid(void);

Returns: effective user ID of calling process

gid_t getgid(void);

Returns: real group ID of calling process

gid_t getegid(void);

Returns: effective group ID of calling process

fork function

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

Function prototype

#include<unistd.h>

pid_t fork(void);

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

The new process created by fork is called the child process

This function is called once but returns twice.

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

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

The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent.

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. 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

Program 1

```
/* Program to demonstrate fork function - Program name – fork1.c */
#include<sys/types.h>
#include<unistd.h>
int main( )
{
        fork( );
        printf("\n hello USP");
}
```

Output : $ cc fork1.c

$ ./a.out

hello USP

hello USP Note : The statement hello USP is executed twice as both the child and parent have executed that instruction.

Program 2

/* Program name – fork2.c */

```
#include<sys/types.h>
#include<unistd.h>
 int main( )
{
printf("\n 7th semester ");
fork( );
printf("\n hello USP");
}
```

Output : $ cc fork1.c

$ ./a.out

7th semester

hello USP

hello USP

Note: The statement 7th semester is executed only once by the parent because it is called before fork and statement hello USP is executed twice by child and parent.

File Sharing

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

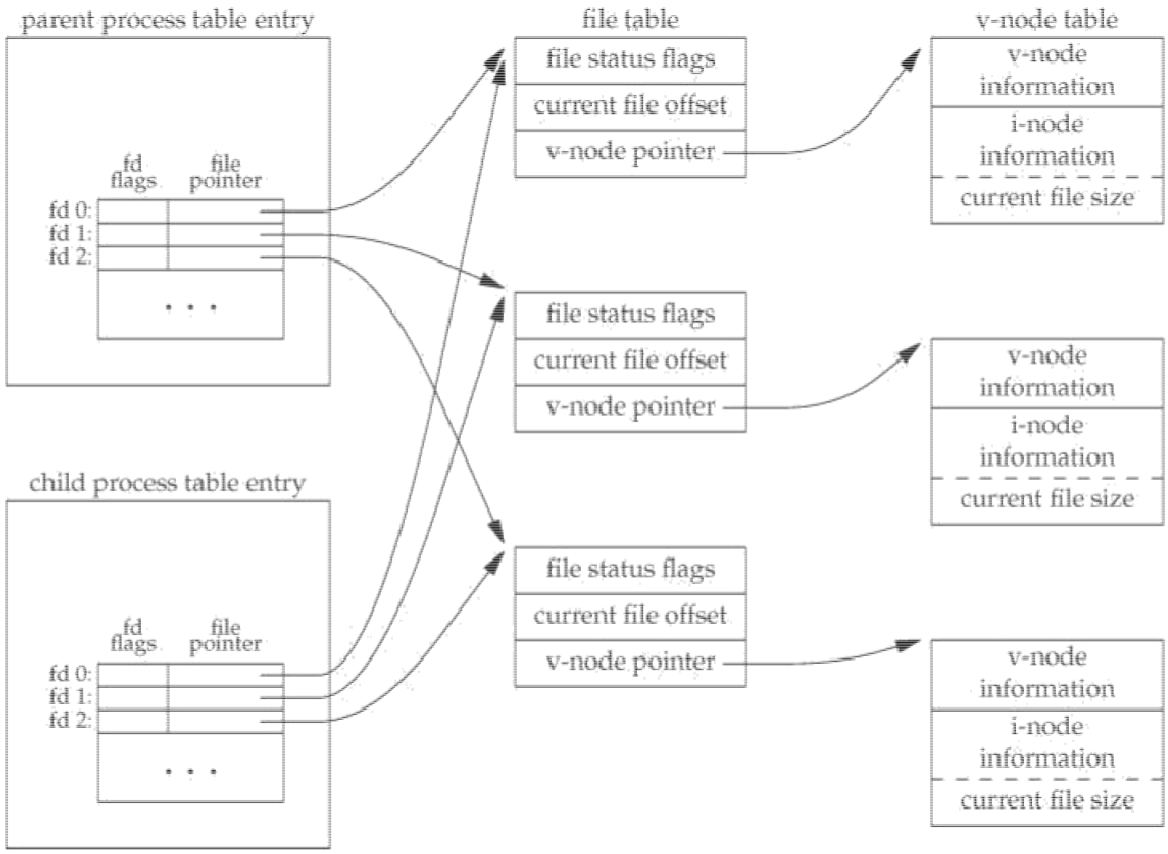


Figure: Sharing of open files between parent and child after fork

It is important that the parent and the child share the same file offset.

Consider a process that forks a child, then waits for the child to complete.

Assume that both processes write to standard output as part of their normal processing. If the parent has its standard output redirected (by a shell), it is essential that the parent's file offset be updated by the child when the child writes to standard output.

In this case, the child can write to standard output while the parent is waiting for it; on completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote.

If the parent and the child did not share the same file offset, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.

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

1. The parent waits for the child to complete.
In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or wrote to, the offsets will be updated accordingly.

2. Both the parent and the child go their own ways.
Here, after the fork, the parent closes the descriptors that it doesn't need, and the child does the same thing. This way, neither interferes with the other's open descriptors. This scenario is often the case with network servers.

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

Real user ID, real group ID, effective user ID, effective group ID
Supplementary group ID
Process group
ID ○ Session ID
Controlling terminal
The set-user-ID and set-group-ID
flags o Current working directory
o Root directory
o File mode creation mask
o Signal mask and dispositions
The close-on-exec flag for any open file
descriptors o Environment list
o Attached shared memory segments
o Memory mappings
o Resource limits

The differences between the parent and child are

- The return value from fork
- The process IDs are different
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change
- The child's tms_utime, tms_stime, tms_cutime, and tms_cstime values are set to 0
- File locks set by the parent are not inherited by the child
- Pending alarms are cleared for the child
- The set of pending signals for the child is set to the empty set

➢ vfork function

The function vfork has the same calling sequence and same return values as fork.

The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.

The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, the child simply calls exec (or exit) right after the vfork.

Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent.

Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes.

Example of vfork function

```
#include "apue.h"
int       glob = 6;          /* external variable in initialized data */
int main(void)
{
    int       var;           /* automatic variable on the stack */
    pid_t    pid;
    var = 88;
    printf("Before vfork\n");
    if ((pid = vfork()) < 0)
    {
        err_sys("vfork error");
```

```
    }
    else if (pid == 0)
    {
        glob++;
        var++;



/* child */
/* modify parent's variables */

        _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
```

- Here the incrementing of a variables is done by child, changes the values in the parent.

➢ 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 also 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. Return immediately with an error, if it doesn't have any child processes.

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

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

The differences between these two functions are as follows.

- The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates.
If the caller blocks and has multiple children, wait returns when one terminates.

For both functions, the argument statloc is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

The interpretation of the pid argument for waitpid depends on its value:

pid == 1	Waits for any child process. In this case, 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.

The waitpid function provides three features that aren't provided by the wait function.

The waitpid function lets us wait for one particular process.

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

The waitpid function provides support for job control with the WUNTRACED and WCONTINUED options.

Macros to examine the termination status returned by wait and waitpid

WIFEXITED(status)	True if status was returned for a child that terminated normally.
WIFSIGNALED (status)	True if status was returned for a child that is terminated abnormally
WIFSTOPPED (status)	True if status was returned for a child that is currently stopped

Program to print a description of the exit status

```
#inculde “apue.h”
#inculde <sys/wait.h>

Void pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status =
        %d\n", WEXITSTATUS(status);


else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number =
%d%s\n", WTERMSIG(status);

    else if (WIFSTOPPED(status))
printf("child stopped, signal number =
%d\n”); WSTOPSIG(status);
}
```

Avoid zombie processes by calling fork twice

```
#include "apue.h"
#include
<sys/wait.h> int
main(void)
{
    pid_t   pid;
    if ((pid = fork()) < 0)
    {
        err_sys("fork error");
    }
    else if (pid == 0) {          /* first child */
        if ((pid = fork()) < 0)
```

```
            err_sys("fork error");
        else if (pid > 0)
            exit(0);      /* parent from second fork == first child */

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

        sleep(2);
        printf("second child, parent pid =
            %d\n",getppid()); exit(0);
    }

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

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

}
```

Output:

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

➢ wait3 and wait4 functions

The only feature provided by these two functions that isn't provided by the wait 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 (Such as read, write…) and the final outcome depends on the order in which the processes run.

For example if two people tried to turn on the light using two different switches at exactly the same time. One instruction might cancel other or two actions might trip the circuit breaker.

If a process wants to wait for a child to terminate, it must call one of the wait function.
If a process wants to wait for its parent to terminate, a following loop can be used.

```
while (getppid ()! =1)
sleep (1);
```

The problem with this type of loop is that it wastes the CPU time, since the caller is woken up every second to test the condition. (Is called polling).

To avoid race condition and polling some form of signalling is required between multiple processes.

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

```
#include “ourhdr.h”
static void charatatime(char *);
```

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

Output:

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

$./a.out
oouuttppuutt ffrroomm cchhppaarrntt

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

Example: Program modification to **avoid race condition** (use the TELL and WAIT functions)

```
#include “ourhdr.h”
    static void charatatime(char *);
```

```
    int main(void)
    {
        pid_t   pid;
+       TELL_WAIT();
+
        if ((pid = fork()) < 0) {
            err_sys("fork error");
        }
        else if (pid == 0) {
+           WAIT_PARENT();
            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 only replaces the current process – (its text, data, heap, and stack segments with a brand new program from disk).

There are 6 exec functions:

#include <unistd.h>

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

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

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

int **execve**(const char *pathname, char *const argv[], char *const

envp[]); int **execlp**(const char *filename, const char *arg0, ... /* (char

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

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

Difference between above functions

1. The first difference in these functions is that the first four functions (execl, execv, execle, execve) take a pathname argument, whereas the last two functions (execlp, execvp) take a filename argument.

   When a filename argument is specified

   i. If filename contains a slash, it is taken as a pathname.

   ii. Otherwise, the executable file is searched 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.

2. The final difference is the passing of the environment list to the new program.

The two functions whose names ends with e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings.

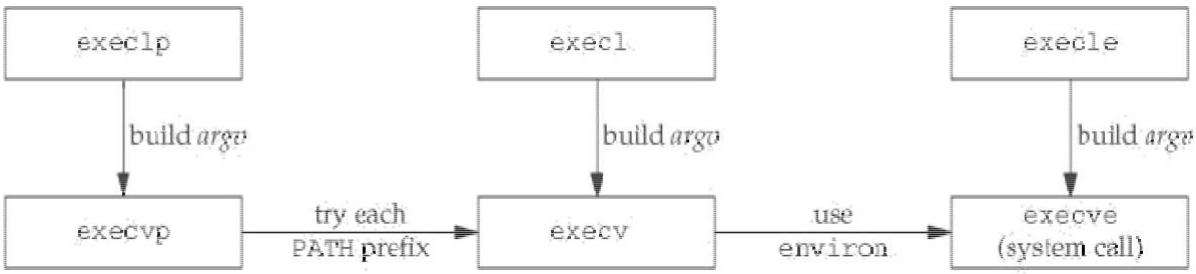
The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.

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:

Process ID and parent process ID
Real user ID and real group ID
Supplementary group IDs
Process group
ID Session ID
Controlling terminal
Time left until alarm clock Current working directory Root directory
Pending signals
File locks
Resource limits
Values for tms_utime, tms_stime, tms_cutime, and tms_cstime.



Relationship of the six exec functions

Example of exec functions

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

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

int main(void)
{
    pid_t   pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    }
    else if (pid == 0) { /* specify pathname, specify environment */ if
     (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                  " myarg2", (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[1]:myarg2
USER = unknown
PATH = /tmp
```

Echo all command-line arguments and all environment strings

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

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

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


    exit(0);
}
```