MODULE-3

Topics

UNIX File APIs:
General File APIs, File and Record Locking, Directory File APIs, Device File APIs, FIFO File APIs, Symbolic Link File APIs.
UNIX Processes and Process Control:
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

CHAPTER 6 UNIX File APIs

Introduction

In UNIX everything can be treated as files. Hence files are building blocks of UNIX operating system. When you execute a command in UNIX, the UNIX kernel fetches the corresponding executable file from a file system, loads its instruction text to memory and creates a process to execute the command.

UNIX / POSIX file Types

The different types of files available in UNIX / POSIX are:

Regular files	Example: All .exe files, C, C++, PDF Document files.
Directory files	Example: Folders in Windows.
Device files	Example: Floppy, CDROM and Printer.
FIFO files	Example: Pipes.
Link Files (only in UNIX)	Example: alias names of a file, Shortcuts in Windows.

Regular files

- It is a text or binary file.
- These files may be read or written by users with the appropriate access permission.
- For Example: All exe files, text, Document files and program files. Created and modified by using editors (vi, vim, emacs) or by compilers (cc, gcc, c++,g++).
- And these files are removed by using *rm* command.

Directory files

- A directory is like a file folder that contains other files, including other subdirectories.

- It provides a means for users to organize their files into some hierarchical structure based on file relationship or users.
- mkdir is the command used for creating the directory.
- mkdir newdir // creates an empty directory newdir under the current directory.
- An unix directory is consider to be an empty if it contains no other files except . and .. files.
- *rmdir* is a command, which is used to delete a directory(s).
- rmdir newdir // To remove the directory that directory should be empty.
- The contents of a directory files may be displayed in UNIX by the *ls* command.

Device files

- These are physical devices such as printers, tapes, floppy devices CDROMs, hard disks and terminals.
- There are two types of device files: Block and Character device files.

a. Block Device files:

- A physical device that transmits block of data at a time. For example: floppy devices CDROMs, hard disks.

b. Character Device files:

- A physical device that transmits data character by character. For example: Line printers, modems etc.
- mknod is the command used for creating the device files.

  Syntax: **$mknod Device_Filename DeviceFile_Type Major_Number Minor_Number**

  Example1: # mknod /dev/cdsk c 120 20

  This creates character device file called *cdsk* with major number 120 and minor number 20.

  Example2: # mknod /dev/bdsk b 225 15

  This creates block device file called *bdsk* with major number 225 and minor number 15.
- A major device number is an index to a kernel table that contains the addresses of all device driver functions known to the system.
- A minor device number is an integer value to be passed as an argument to a device driver function when it is called.
- The minor device number tells the device driver function what actual physical device it is taking to and the input output buffering scheme to be used for data transfer.
- In UNIX, *mknod* must be invoked through superuser privileges.

FIFO file

- It is a special pipe device file, which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer.
- A process may write more than PIPE_BUF bytes of data to a FIFO file, but it may be blocked when the file buffer is filled.
- In this case the process must wait for a reader process to read data from the pipe and make room for the write operation to complete.

- Finally, the data in the buffer is accessed in a by a first-in-first-out manner, hence the file is called a FIFO.
- These are having name hence they called named pipes.
- The size of the buffer associated with FIFO file is fixed to PIPE_BUF or _POSIX_PIPE_BUF.
- *mkfifo* is the command used for creating a FIFO file(s).

  mkfifo /usr/prog/fifo_pipe **or** by using *mknod* with the option *p*

  mknod /usr/prog/fifo_pipe p
- *rm* is the command used for removing FIFO files.

Symbolic Link files

- A symbolic link file contains a path name which references another file in either the local or a remote file system.
- These are not supported by POSIX.1
- A symbolic link may be created in UNIX via the *ln* command.
- This command creates a symbolic link /usr/csr/unix which references the file /usr/yuktha/original. The cat command which follows will print the content of the /usr/yuktha/original file:
- Example: ln –s /usr/yuktha/original /usr/csr/unix cat -n /usr/csr/unix
- It is possible to create a symbolic link to reference another symbolic link.
- rm, mv and chmod commands will operate only on the symbolic link arguments directly and not on the files that they reference.

6.1 General File APIs

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

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

Open:

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

Prototype:

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

- **path_name :** The pathname of a file to be opened or created. It can be an absolute path name or relative path name. The pathname can also be a symbolic link name.
- **access_mode:** An integer values in the form of manifested constants which specifies how the file is to be accessed by calling process. The manifested constants can be classified as access mode flags and access modifier flags.

Access mode flags:

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

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

Access Modifier Flags:

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

  Example:

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

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

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

Permission:

- The permission argument is required only if the O_CREAT flag is set in the access_mode argument. It specifies the access permission of the file for its owner, group and all the other people.
- Its data type is *int* and its value is octal integer value, such as 0764. The left-most, middle and right-most bits specify the access permission for owner, group and others respectively.
- In each octal digit the left-most, middle and right-most bits specify read, write and execute permission respectively.
- For example 0764 specifies 7 is for owner, 6 is for group and 4 is for other.
  7 = 111 specifies read, write and execution permission for owner.
  6 = 110 specifies read, write permission for group.
  4 = 100 specifies read permission for others.
  Each bit is either 1, which means a right is granted or zero, for no such rights.
- POSIX.1 defines the permission data type as mode_t and its value is manifested constants which are aliases to octal integer values. For example, 0764 permission value should be specified as:
  S_IRWXU|S_IRGRP|S_IWGRP|S_IROTH
- *Permission* value is modified by its calling process *umask* value. An *umask* value specifies some access rights to be masked off (or taken away) automatically on any files created by process.
- The function prototype of the *umask* API is:
  mode_t umask (mode_t new_umask);
  It takes new mask value as argument, which is used by calling process and the function returns the old umask value. For example,
  mode_t old_mask = umask (S_IXGRP | S_IWOTH |S_IXOTH);
  The above function sets the new umask value to "no execute for group" and "no write-execute for others".
- The open function takes its permission argument value and bitwise-ANDs it with the one's complement of the calling process umask value. Thus,
  actual_permission = permission & ~umask_value
  Example: **actual_permission = 0557 & (~031) = 0546**
  The return value of open function is -1 if the API fails and errno contains an error status value. If the API succeeds, the return value is file descriptor that can be used to reference the file and its value should be between 0 and OPEN_MAX-1.

Creat:

- The creat system call is used to create new regular files.

  Prototype:

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

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

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

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

read:

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

  Prototype:

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

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

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

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

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

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

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

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

write:

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

Its prototype is:

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

- **fdesc:** is an integer file descriptor that refers to an opened file.
- **buf:** is the address of a buffer which contains data to be written to the file.
- **size:** specifies how many bytes of data are in the buf argument.

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

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

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

close:

- The close function disconnects a file from a process.

Prototype:

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

- **fdesc:** is an integer file descriptor that refers to an opened file.
- The return value of close is zero if the call succeeds or -1 if it fails.

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

fcntl:

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

  Prototype:

  #include <fcntl.h>

  int fcntl (**int** fdesc ,**int** cmd, ….);

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

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

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

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

  int cur_flags = fcntl(fdesc, FGETFL);

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

- The close-on-exec flag of a file descriptor specifies that if the process that owns the descriptor calls the exec API to execute different program, the fdesc should be closed by the kernel before the new program runs or not.
- The example reports the close-on-exec flag of a fdesc, sets it to on afterwards:
  cout<<fdesc<<"close-on-exec:"<<fcntl(fdesc,F_GETFD)<<endl;
  (void)fcntl(fdesc, F_SETFD, 1);
- The fcntl function can also be used to duplicate a fdesc with another fdesc. The results are two fdesc reference the same file with same access mode and share the same file pointer to read or write the file. This is useful in the redirection of standard input or output to reference a file.
  Example: Reference standard input of a process to a file called FOO

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

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

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

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

lseek:

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

  Prototype:

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

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

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

***NOTE:**

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

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

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

- The return value of lseek is the new file offset address where the next read of write operation will occur, or -1 if lseek call fails.
- The iostream class defines *tellg* and *seekg* functions to allow users to randomly access data from any isotream class.
- These functions can be implemented using the *lseek* function as follows:

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

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

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

link:

- The link function creates a new link for an existing file . This function does not create a new file. It create a new path name for an existing file. Its prototype is:
  #include <unistd.h>

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

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

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

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

unlink:

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

  Prototype:

  #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.

Prototype:

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

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

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

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

stat, fstat:

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

The prototype is given below:

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

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

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

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

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

Prototype:

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

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

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

static char xtbl[10] = "rwxrwxrwx";

static void display_file_type ( ostream& ofs, int st_mode )
{
   switch (st_mode &S_IFMT)
   {
        case S_IFDIR:    ofs << 'd'; return;          /* directory file */
        case S_IFCHR:    ofs << 'c'; return;          /* character device file */
        case S_IFBLK:    ofs << 'b'; return;          /* block device file */
        case S_IFREG:    ofs << ' '; return;          /* regular file */
        case S_IFLNK:    ofs << 'l'; return;          /* symbolic link file */
        case S_IFIFO:    ofs << 'p'; return;          /* FIFO file */
   }
}
/* Show access permission for owner, group, others, and any special flags */
static void display_access_perm ( ostream& ofs, int st_mode )
{
        char amode[10];
        for (int i=0, j= (1 << 8); i < 9; i++, j>>=1)
        amode[i] = (st_mode&j) ? xtbl[i] : '-';          /* set access permission */
```

```
        if (st_mode&S_ISUID) amode[2] = (amode[2]=='x') ? 'S' : 's';
        if (st_mode&S_ISGID) amode[5] = (amode[5]=='x') ? 'G' : 'g';
        if (st_mode&S_ISVTX) amode[8] = (amode[8]=='x') ? 'T' : 't';
        ofs << amode << ' ';
}
/* List attributes of one file */
static void long_list (ostream& ofs, char* path_name)
{
    struct stat         statv;
    struct group*gr_p;
    struct passwd*pw_p;
    if (lstat (path_name, &statv))
    {
        cerr<<"Invalid path name:"<< path_name<<endl;
        return;
     }
    display_file_type( ofs, statv.st_mode );
    display_access_perm( ofs, statv.st_mode );
    ofs << statv.st_nlink;                  /* display hard link count */
    gr_p = getgrgid(statv.st_gid);          /* convert GID to group name */
    pw_p = getpwuid(statv.st_uid);          /*convert UID to user name */

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

access:

The access function checks the existence and/or access permission of user to a named file.

Prototype:

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

path_name: The pathname of a file.

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

Bit Flag	Use

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

The flag argument value to access call is composed by bitwise-ORing one or more of the above bit-flags. The following statement checks whether a user has read and write permissions on a file /usr/sjb/file1.doc:
int rc = access("/usr/sjb/file1.doc", R_OK|W_OK);

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

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

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

chmod, fchmod:

The chmod and fcmod functions change file access permissions for owner, group and others and also set-UID, set-GID and sticky flags.
A process that calls one of these functions should have the effective user ID of either the super user or the owner of the file.

The prototype of these functions is given below:

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

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

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

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

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

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

chown, fchown, lchown:

- The chown and fchown functions change the user ID and group ID of files. They differ only in their first arguments 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);
```

  - 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.

In BSD UNIX:

- A process with super user privilege can use these functions to change any file user and group ID.
- If a process effective user ID matches a file user ID and its effective group ID or one of its supplementary group IDs match the file group ID, the process can change the file group ID only.

In UNIX System V:

- A process whose effective user ID matches either the user ID of a file or the user ID of a super user can change the file user ID and group ID.

In POSIX.1:

- Specifies that if the _POSIX_CHOWN_RESTRICTED variable is defined with a non -1 value, *chown* should behave as in BSD UNIX.
- If the _POSIX_CHOWN_RESTRICTED variable is undefined, *chown* should behave as in UNIX System V.

The following *test_chown.C* program implements the UNIX *chown* program:

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

int main (int argc, char" argv0)
 {
   if(argc<3) {
     cerr << "Usage: " << argv[0] << " <usr_name> <file> ...\n"; return 1;
   }
  }
  struct passwd *pwd = getpwuid( argv(1 ] );    /* r convert user name to UID **/
  uid t             UID = pwd ? pwd->pw_uid : -1;
  struct stat       state:
  if (UID == (uid_t)-1)
     cerr<< "Invalid user name\n";
  else for (int i=2; i < argc; i++)        /* do for each file specified *r
  if (stat(argv[i],&state)==o) {
        if (chown( argv(i], UID, staty.st_gid) )
           perror( "chown" );
     } else perror( "stat" );
  return 0;
}
```

This program takes, at least two command line arguments: the first one is a user name to be assigned to files, and the second and any subsequent arguments are file path names. The program

first converts a given user name to a user ID via the *getpwuid* function. If this succeeds, the program processes each named file as follows: it calls *stat to* get the file group ID, then calls *cltown* to change the file user ID. If either the stator *chown* API fails, *perror* will be called to print a diagnostic message.

utime:

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

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

The *path_name* argument is the path name of a file.
The *times* argument specifies the new access time and modification time for the file.
The *struct utimbuf* is defined in the <utime.h> header as:

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

If *times* is specified as 0, the API will set e named file access time and modification time to the current time. This requires that the calling process have write access to the named file, its effective user ID match either the file user ID or that of the super user.

The return value of *utime* is 0 if it succeeds or -1 if it fails. Possible failures of the API may be: The *path_name* argument is invalid, the process has no access permission and ownership to a named file, or the *times* argument has an invalid address.

The following *test_touch.C* program uses the *utime* function to change the access and modification time stamps of files. The time stamp to set is also defined by users:

```
/* Usage: a.out <offset in seconds> <file> ... */
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <utime.h>
#include <time.h>

int main (int argc, char* argv[] ) {
   struct utimbuf times;
   int            offset;

    if (argc < 3 II sscanf( argv[1], "%d", &offset) I= 1) {
      cerr << "usage:" << argv[0] << " <offset> <file> _An"; return 1;
    }

    /* new time is current time + offset in seconds */ times.actime =
    times.modtime = time( 0) + offset;
```

```
    for ( -i=1; i < argc; i ++)          /* touch each named file */

    if ( utime ( argv[i], &times)) perror( "utime" );
    return 0;
}
```

6.2 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*.

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.

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:
  1. 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.
  2. After a lock is acquired successfully, read or write the locked region release the lock
- The drawback of advisory locks is that programs that create processes to share files must follow the above file locking procedure to be cooperative. This may be difficult to control when programs are obtained from different sources.
- All UNIX and POSIX systems support advisory locks.

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

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

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

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

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

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

The possible values of l_type are:

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

The possible values of l_whence and their uses are:

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

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:

- Tum on the set-GID flag of the file.
- Tum off the group execute right 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, FSETLK,&fvar)==-1) {
        /* Set lock falls, find out who has locked the file */
        while (fcntl(fdesc,F_GETLK,&fvar)!=-1 && fvar.l_type != F_UNLCK){
        cout<<*argv<<"locked by"<<fvar.l_pid<<"from"<<fvar.l_start<<"for"<<fvar.l_len
             <<"byte for"<<(fvar.l_type == F_WRLCK ? 'w':'r')<<endl;
           if (!fvar.l_len) break;
           fvar.l_start += fvar.l_len;
            fvar.l_len   = 0;
           }/* while there are locks set by other processes */
       } /* while set lock un-successful */
```

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

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

6.3 Directory File APIs

- Directory files in UNIX and POSIX systems are used to help users in organizing their files into some structure based on the specific use of file.
- They are also used by the operating system to convert file path names to their inode numbers.
- Directory files are created in BSD UNIX and POSIX.1 by mkdir API:
  - #include <sys/stat.h>
  - #include <unistd.h>
  - int *mkdir (* **const char*** path_name, **mode** t mode);

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

- UNIX System V.3 uses the *mknod* API to create directory files.
- UNIX System V.4 supports both the *mkdir* and *mknod APIs* for creating directory files.
- The difference between the two *APIs is* that a directory created by ***mknod*** **does not contain the "."** and ".." links. On the other hand, a directory created by *mkdir* has the "." and ".." links created in one atomic operation, and it is ready to be used.
- A directory file is a record-oriented file, where each record stores a file name and the mode number of a file that resides in that directory.
- The following portable functions are defined for directory file browsing. These functions are defined in both the <dirent.h> and <sys/dir.h> headers.

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

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

- The uses of these functions are:

  opendir: Opens a directory file for read-only. Returns a file handle DIR* for future reference of the file.

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

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

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

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

  telldir: Returns the file pointer of a given dir_fdesc.

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

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

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

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

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

int main (int argc, char* argv[ ])
 {
  Dirent*          dp;
  DIR*  dir_fdesc;
  while (--argc > 0 ) {       /* do the following for each file */
  if ( !(dir_fdesc = opendir( *++argv ) )) {
        if (mkdir( *argv, S_ IRWXU|S_IRWXG|S_IRWXO) == -1 )
         perror( "opendir" );
        continue;
     }
```

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

6.4 Device File APIs

- Device files are used to interface physical devices with application programs.
- Specifically, when a process reads or writes to a device file, the kernel uses the major and minor device numbers of a file to select a device driver function to carry out the actual data transfer.
- Device files may be character-based or block-based.
- UNIX systems define the ***mknod*** API **to create device** files.

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

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

6.5 FIFO File APIs

- FIFO files are also known as named pipes.
- They are special pipe device files used for inter process communication. Any process can attach to a FIFO file to read, write, or read-write data.
- Data written to a FIFO file are stored in a fixed-size (PIPE BUF) buffer and are retrieved in a FIFO order.

BSD UNIX and POSIX.1 define the *mkfifo* API to create FIFO files:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int mkfifo ( 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, as well as the S_IFIFO flag to indicate that this is a FIFO file.
3. For example, to create a FIFO file called *FIFO5* with access permission of read-write-execute for everyone, the *mkfifo* call is:

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

- When a process opens a FIFO file for read-only, the kernel will block the process until there is another process that opens the same file for write.

- Similarly, if a process opens a FIFO for write, it will be blocked until another process opens the FIFO file for read. This provides a method for processes synchronization.
- Another special thing about FIFO files is that if a process writes to a FIFO file that has no other process attached to it for read, the kernel will send a SIGPIPE signal to the process to notify it of the illegal operation.
- If a process does not desire to be blocked by a, FIFO file, it can specify the O_NONBLOCK flag in the *open* call to the-FIFO file. With this flag, the *open* API will not block the process even though there is no process attached to the other end of the FIFO file.

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

```
#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>]\n";
      return 0;
  }
  int      fd;
  char    but [256];
  if (argue=2) {                          /' reader process * /
  (void)mkfifo( argv[1], S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO );
  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 {                                 /* writer process' */ fd = open( argv[1], O_WRONLY
       );
       write( fd, argv[2], strlen( argv[2] ) );
 }
 close(fd);
}
```

6.6 Symbolic Link API:

- Symbolic links are developed to overcome several shortcomings 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.
- Symbolic links are being proposed to be included the POSIX.1 standard. BSD UNIX defines the following APIs for symbolic links manipulation:

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

symlink Function

- The *org_link* and *sym_link* arguments to a *symlink* call specify the original file path name and the symbolic link path name to be created.
- For example, to create a symbolic-link called */usr/joe/lnk* for a file called */usr/go/testl*, the *symlink* call will be:

  symlink ("/usr/go/testl", "usr/roe/Ink");

- The return value of *symlink* is 0 if it succeeds or -1 if it fails, Possible causes of failure are: The path name specified is illegal, the *sym_link* file already exists, or the calling process lacks permission to create the new file.

readlink Function

- To query the path name to which a symbolic link refers, users must use the *readlink* API.
- The arguments to the *readlink* API are: *sym_link* is the path name of a symbolic link, *buf* is a character array buffer that holds the return path name referenced by the link, and *size* specifies the maximum capacity (in number of bytes) of the *buf* argument.
- The return value of *readlink is -1* if it fails or the actual number of characters of a path name that is placed in the *buf* argument.

  The following function takes a symbolic link path name as argument, and it will call *readlink* repeatedly to resolve all links to the file. The *while* loop terminates when *readlink* returns -1, and the *buf* variable contains the nonlink file path name, which is then printed to the standard output:

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

int resolve_link( const char* sym_link )
{
   char* buf[256], tname[256];
   strcpy(tname,sym_link);
   while ( readlink( tname, buf, sizeof( buf)) > 0 )
   strcpy( tname, buf );
   cout <<sym_link << " => " <<buf << endl;
}
```

lstat Function

- The *lstat* function is used to query the file attributes of symbolic links.
- The UNIX Is -1 command uses *lstat* to display information of all file type, including symbolic links.

The following *test_symin.C* program emulates the UNIX In command. The main function of the program is to create a link to a file. The names of the original file and new link are specified as the arguments to the program, and if the -s option is not specified, the program will create a hard link. Otherwise, it will create a symbolic link:

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>
 #include <string.h>
/* Emulate the UNIX In command */
 int main (int argc, char* argv[ ])
{
    char* buf[256], tname[256];
    if ((argc< 3 && argc > 4) || (argc==4 && strcmp(argv[1 ];'-s")))
    {
     cout << "usage: " << argv[0] << " [-s] <orig file> <new_link>\n";
     return 1;
    }
    if (argc==4)
       return symlink( argv[2], argv[3]); /* create a symbolic link */
       else
       return link(argv[1], argv[2]); /* create a hard link */
}
```

UNIX PROCESSES AND PROCESS CONTROL

Chapter 7 The Environment of a UNIX Process

7.1 Introduction

- Before looking at the process control primitives, we need to examine the environment of a single process.
- We'll 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 various ways for the process to terminate.
- Additionally, we'll look at the longjmp and setjmp functions and their interaction with the stack.
- Lastly we see the resource limits of a process.

7.2 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,
  → 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.

7.3 Process Termination

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

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

Abnormal termination occurs in three ways:

6. Calling abort

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

The start-up routine is also written so that if the main function returns, the exit function is called. If the start-up routine were coded in C (it is often coded in assembler) the call to main could look like

```
exit(main(argc, argv));
```

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);
```

- The reason for the different headers is that exit and _Exit are specified by ISO C, whereas _exit is specified by POSIX.1.
- Historically, the exit function has always performed a clean shutdown of the standard I/O library: the fclose function is called for all open streams. This causes all buffered output data to be flushed (written to the file).
- All three exit functions expect a single integer argument, which we call the exit status. Most UNIX System shells provide a way to examine the exit status of a process. If
  (a) Any of these functions is called without an exit status,
  (b) main does a return without a return value, or
  (c) The main function is not declared to return an integer,
- The exit status of the process is undefined.
- However, if the return type of main is an integer and main "falls off the end" (an implicit return), the exit status of the process is 0.
- This behavior is new with the 1999 version of the ISO C standard. Historically, the exit status was undefined if the end of the main function was reached without an explicit return statement or call to the exit function.
- 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.

Example

The program below is the classic "hello, world" example.

```
#include <stdio.h>
main()
{
        printf("hello, world\n");
}
```

- When we compile and run this program, we see that the exit code is random. If we compile the same program on different systems, we are likely to get different exit codes, depending on the contents of the stack and register contents at the time that the main function returns:

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

- Now if we enable the 1999 ISO C compiler extensions, we see that the exit code changes:

```
$ cc -std=c99 hello.c      enable gcc's 1999 ISO C extensions
hello.c:4: warning: return type defaults to 'int'
$ ./a.out
hello, world
$ echo $?              print the exit status
0
```

- Note the compiler warning when we enable the 1999 ISO C extensions. This warning is printed because the type of the main function is not explicitly declared to be an integer.

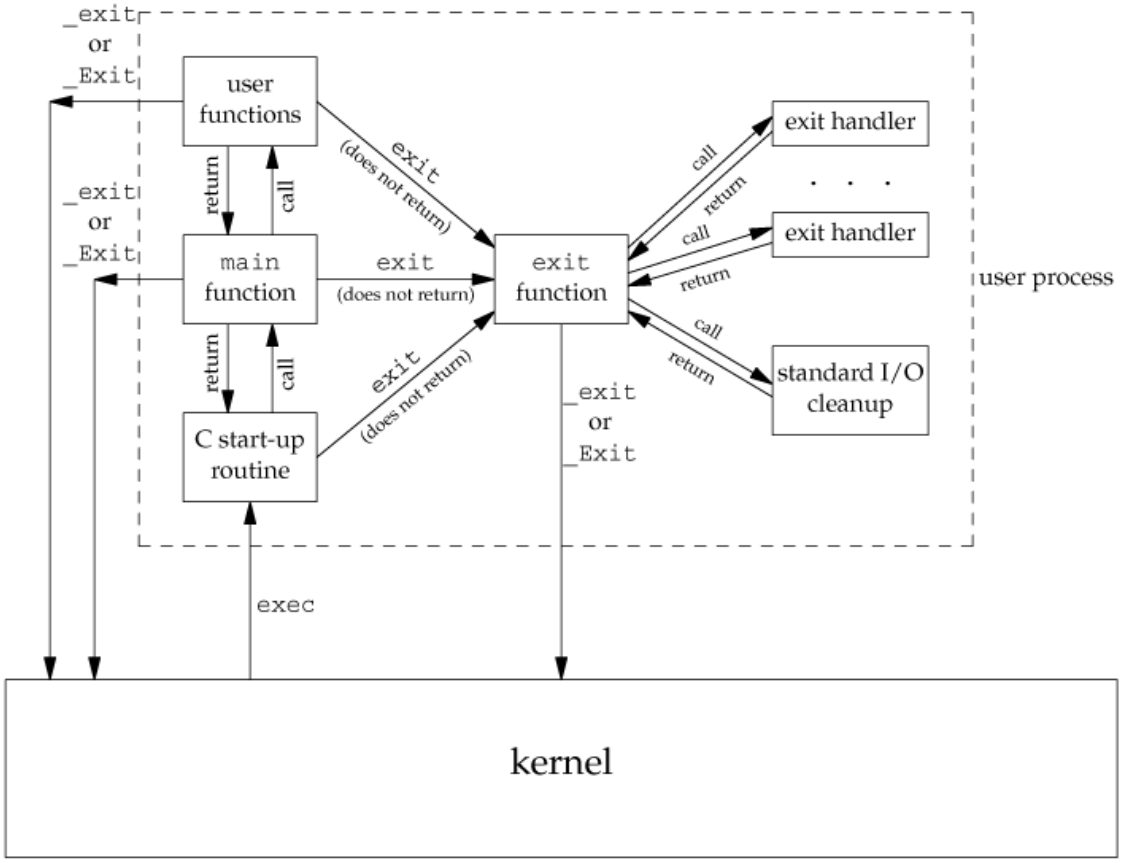
atexit Function

- 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.
- These exit handlers first appeared in the ANSI C Standard in 1989. Systems that predate ANSI C, such as SVR3 and 4.3BSD, did not provide these exit handlers.
- ISO C requires that systems support at least 32 exit handlers. The sysconf function can be used to determine the maximum number of exit handlers supported by a given platform.
- With ISO C and POSIX.1, exit first calls the exit handlers and then closes (via fclose) all open streams.
- POSIX.1 extends the ISO C standard by specifying that any exit handlers installed will be cleared if the program calls any of the exec family of functions.

Figure below summarizes how a C program is started and the various ways it can terminate.



- Note that the only way a program is executed by the kernel is when one of the exec functions is called. The only way a process voluntarily terminates is when _exit or _Exit is called, either explicitly or implicitly (by calling exit). A process can also be involuntarily terminated by a signal (not shown in Figure).

Example

1. The program in below demonstrates the use of the atexit function.

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

static void my_exit1(void);
static void my_exit2(void);

int main(void)
{
    if (atexit(my_exit2) != 0){
      printf("can't register my_exit2");
    exit(1);
    }
```

```
    if (atexit(my_exit1) != 0){
      printf("can't register my_exit1");
    exit(1);
    }

    if (atexit(my_exit1) != 0){
      printf("can't register my_exit1");
    exit(1);
    }

    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");

}
```

Executing this program in yields

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

An exit handler is called once for each time it is registered. In the program above, the first exit handler is registered twice, so it is called two times. Note that we don't call exit; instead, we return from main.

7.4 Command-Line Arguments

- When a program is executed, the process that does the exec can pass command-line arguments to the new program.
- This is part of the normal operation of the UNIX system shells.

Example

The program below echoes all its command-line arguments to standard output.

```
#include <stdio.h>
#include <stdlib.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);
}
```

If we compile this program and name the executable echoarg, we have

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

We are guaranteed by both ISO C and POSIX.1 that argv[argc] is a null pointer. This lets us alternatively code the argument-processing loop as

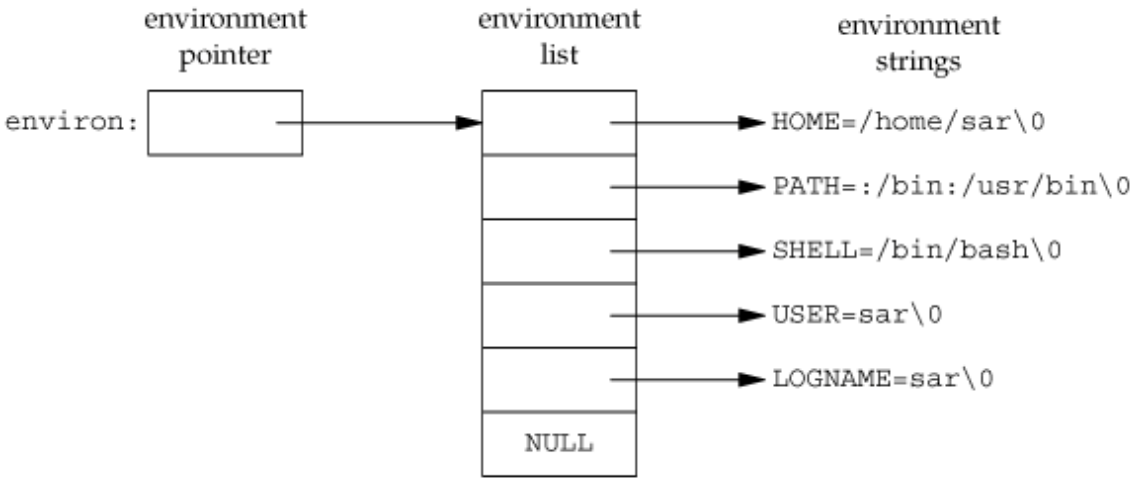
```
for (i = 0; argv[i] != NULL; i++)
```

7.5 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;
```

For example, if the environment consisted of five strings, it could look like as shown in figure below.



- By convention, the environment consists of name=value strings, as shown in figure above. Most predefined names are entirely uppercase, but this is only a convention.
- Historically, most UNIX systems have provided a third argument to the main function that is the address of the environment list:
  - int main(int argc, char *argv[], char *envp[]);
- Because ISO C specifies that the main function be written with two arguments, and because this third argument provides no benefit over the global variable environ, POSIX.1 specifies that environ should be used instead of the (possible) third argument. Access to specific environment variables is normally through the getenv and putenv functions, instead of through the environ variable. But to go through the entire environment, the environ pointer must be used.

7.6 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

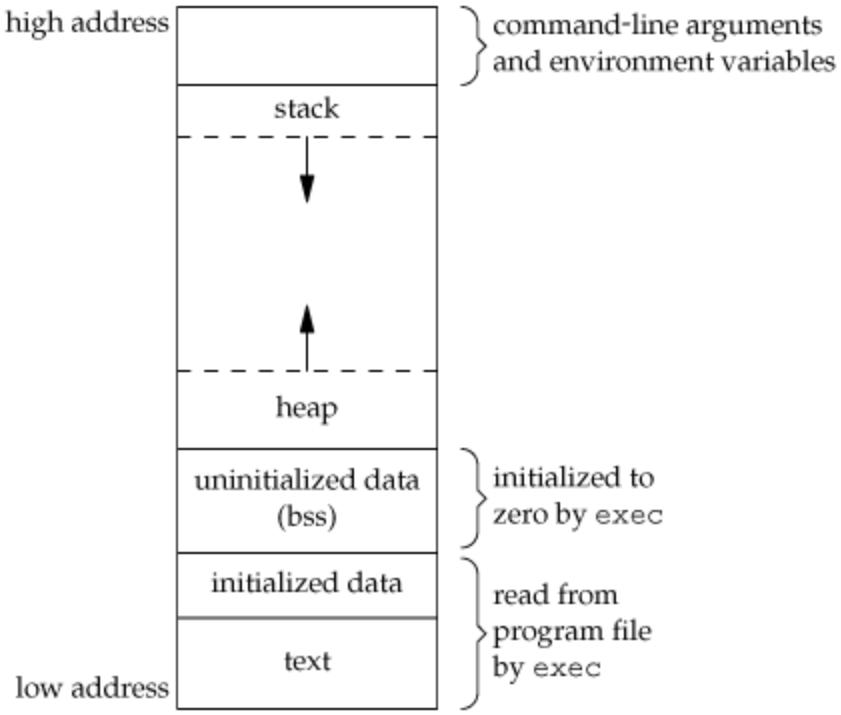
  long sum[1000];

  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.

The figure below shows the typical arrangement of these segments.



- This is a logical picture of how a program looks; there is no requirement that a given implementation arrange its memory in this fashion. Nevertheless, this gives us a typical arrangement to describe.

- With Linux on an Intel x86 processor, the text segment starts at location 0x08048000, and the bottom of the stack starts just below 0xC0000000. (The stack grows from higher-numbered addresses to lower-numbered addresses on this particular architecture.) The unused virtual address space between the top of the heap and the top of the stack is large.
- Several more segment types exist in an a.out, containing the symbol table, debugging information, linkage tables for dynamic shared libraries, and the like. These additional sections don't get loaded as part of the program's image executed by a process.
- Note from then figure above that the contents of the uninitialized data segment are not stored in the program file on disk.
- This is because the kernel sets it to 0 before the program starts running. The only portions of the program that need to be saved in the program file are the text segment and the initialized data.
- The **size** command reports the sizes (in bytes) of the text, data, and bss segments.
  For example:

```
$ size /usr/bin/cc /bin/sh
   text    data     bss     dec     hex filename
  79606    1536     916   82058    1408a /usr/bin/cc
 619234   21120   18260  658614    a0cb6 /bin/sh
```

  The fourth and fifth columns are the total of the three sizes, displayed in decimal and hexadecimal, respectively.

7.7 Shared Libraries

- Most UNIX systems today 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 relink edit every program that uses the library. (This assumes that the number and type of arguments haven't changed.)
- Different systems provide different ways for a program to say that it wants to use or not use the shared libraries.
- Options for the **cc** and **ld** commands are typical. As an example of the size differences, the following executable file—the classic hello.c program—was first created without shared libraries:

```
$ cc -static hello1.c          prevent gcc from using shared libraries
$ size a.out
   text    data     bss     dec     hex filename
 375657    3780    3220  382657   5d6c1 a.out
```

- If we compile this program to use shared libraries, the text and data sizes of the executable file are greatly decreased:

```
$ cc hello1.c                gcc defaults to use shared libraries
$ size a.out
   text    data     bss     dec     hex filename
    872     256       4    1132     46c a.out
```

7.8 Memory Allocation

ISO C specifies three functions for memory allocation:

a. ***malloc***, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
b. ***calloc***, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
c. ***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. For example, if the most restrictive alignment requirement on a particular system requires that doubles must start at memory locations that are multiples of 8, then all pointers returned by these three functions would be so aligned.
- The allocation routines are usually implemented with the ***sbrk*** system call. This system call expands (or contracts) the heap of the process.
- 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.
- Other possible errors that can be fatal are freeing a block that was already freed and calling free with a pointer that was not obtained from one of the three alloc functions.
- If a process calls malloc, but forgets to call free, its memory usage continually increases; **this is called leakage.** By not calling free to return unused space, the size of a process's address space

slowly increases until no free space is left. During this time, performance can degrade from excess paging overhead.

- 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.

Alternate Memory Allocators

Many replacements for malloc and free are available. We discuss some of the alternatives here.

a. **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.

b. **Vmalloc:** Vo, Kiem-Phong [1996] 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.

c. **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. Weinstock and Wulf [1988] describe the algorithm, which is based on splitting up memory into buffers of various sizes and maintaining unused buffers on different free lists, depending on the size of the buffers. Free implementations of malloc and free based on quick-fit are readily available from several FTP sites.

d. **alloca Function:** he 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. Nevertheless, many software packages use it, and implementations exist for a wide variety of systems.

7.9. 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.

ISO C defines a function that we can use to fetch values from the environment, but this standard says that the contents of the environment are implementation defined.

```
#include <stdlib.h>
char *getenv(const char *name);
```
Returns: pointer to value associated with name, NULL if not found

Table below shows the functions that are supported by the various standards and implementations.

Function	ISO C	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
getenv	•	•	•	•	•	•
putenv		XSI	•	•	•	•
setenv		•	•	•	•	
unsetenv		•	•	•	•	
clearenv				•		

Note: clearenv is not part of the Single UNIX Specification. It is used to remove all entries from the environment list.

The prototypes for the middle three functions listed in table above 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 operation of these three functions is as follows.

a. 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.

b. 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.

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

7.10 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.

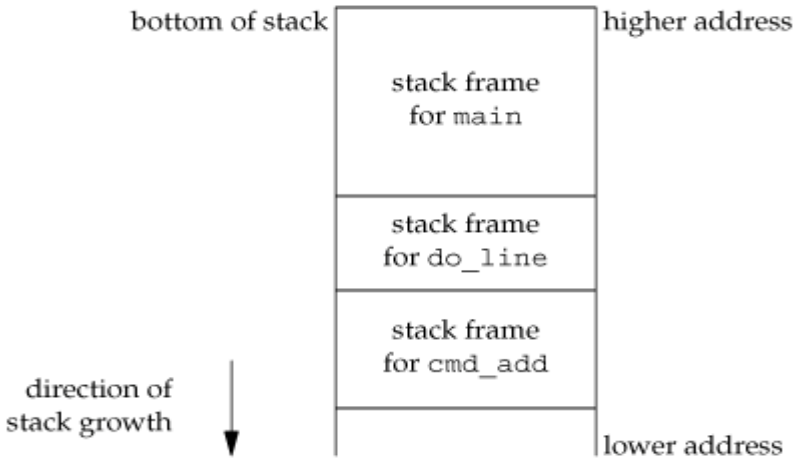
Consider the skeleton of a program as shown below:

```
#include <stdio.h>

#define TOK_ADD    5

void     do_line(char *);
void     cmd_add(void);
int      get_token(void);

int main(void)
{
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
       do_line(line);
    exit(0);
}

char     *tok_ptr;       /* global pointer for get_token() */

void do_line(char *ptr)       /* process one line of input */
{
   int    cmd;

   tok_ptr = ptr;
   while ((cmd = get_token()) > 0) {
      switch (cmd) { /* one case for each command */
      case TOK_ADD:
            cmd_add();
            break;
      }
   }
}

void cmd_add(void)
{
  int      token;

  token = get_token();
  /* rest of processing for this command */
}

int get_token(void)
{
  /* fetch next token from line pointed to by tok_ptr */
```

}

It consists of a main loop that reads lines from standard input and calls the function do_line to process each line. This function then calls get_token to fetch the next token from the input line. The first token of a line is assumed to be a command of some form, and a switch statement selects each command. For the single command shown, the function cmd_add is called.

Figure below shows what the stack could look like after cmd_add has been called.



- Storage for the automatic variables is within the stack frame for each function. The array *line* is in the stack frame for main, the integer *cmd* is in the stack frame for do_line, and the integer *token* is in the stack frame for cmd_add.
- The coding problem that's often encountered with programs like the one shown above is how to handle nonfatal errors.
- For example, if the cmd_add function encounters an error—say, an invalid number—it might want to print an error, ignore the rest of the input line, and return to the main function to read the next input line.
- But when we're deeply nested numerous levels down from the main function, this is difficult to do in C.
- In this example, in the cmd_add function, we're only two levels down from main, but it's not uncommon to be five or more levels down from where we want to return to. It becomes messy if we have to code each function with a special return value that tells it to return one level.
- The solution to this problem is to use the setjmp and longjmp functions.

The prototypes of the two functions are as:

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

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

```
void longjmp(jmp_buf env, int val);
```

- We call setjmp from the location that we want to return to, which in this example is in the main function. In this case, setjmp returns 0 because we called it directly. In the call to setjmp, the argument env is of the special type jmp_buf. This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp. Normally, the env variable is a global variable, since we'll need to reference it from another function.
- When we encounter an error—say, in the cmd_add function—we call longjmp with two arguments. The first is the same env that we used in a call to setjmp, and the second, val, is a nonzero value that becomes the return value from setjmp.
- The reason for the second argument is to allow us to have more than one longjmp for each setjmp. For example, we could longjmp from cmd_add with a val of 1 and also call longjmp from get_token with a val of 2. In the main function, the return value from setjmp is either 1 or 2, and we can test this value, if we want, and determine whether the longjmp was from cmd_add or get_token.
- The program below shows the modification in main and cmd_add functions. The other two functions, do_line and get_token, haven't changed.

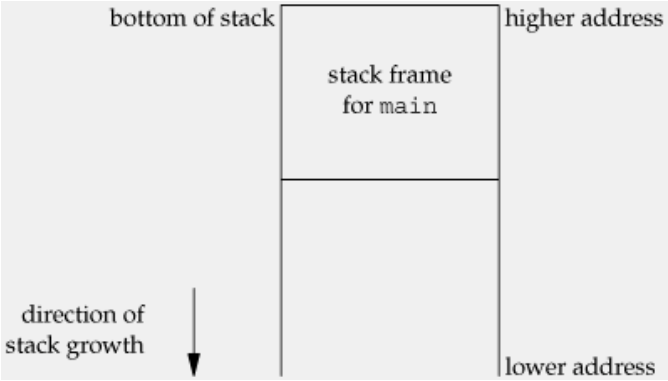
```
#include <stdio.h>
#include <setjmp.h>

#define TOK_ADD    5

jmp_buf jmpbuffer;

int main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
      do_line(line);
    exit(0);
}
 ...

void cmd_add(void)
{
   int     token;

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

- When main is executed, we call setjmp, which records whatever information it needs to in the variable jmpbuffer and returns 0. We then call do_line, which calls cmd_add, and assume that an error of some form is detected.

- Before the call to longjmp in cmd_add, the stack looks like that in figure above. But longjmp causes the stack to be "unwound" back to the main function, throwing away the stack frames for cmd_add and do_line (figure below). Calling longjmp causes the setjmp in main to return, but this time it returns with a value of 1 (the second argument for longjmp).

Automatic, Register, and Volatile Variables

- We've seen what the stack looks like after calling longjmp. The next question is, "what are the states of the automatic variables and register variables in the main function?"
- When main is returned to by the longjmp, do these variables have values corresponding to when the setjmp was previously called (i.e., are their values rolled back), or are their values left alone so that their values are whatever they were when do_line was called (which caused cmd_add to be called, which caused longjmp to be called)?
- Unfortunately, the answer is "it depends." Most implementations do not try to roll back these automatic variables and register variables, but the standards say only that their values are indeterminate. If you have an automatic variable that you don't want rolled back, define it with the volatile attribute. Variables that are declared global or static are left alone when longjmp is executed.



Example

The program below demonstrates the different behavior that can be seen with automatic, global, register, static, and volatile variables after calling longjmp.

```
#include <stdio.h>
#include <setjmp.h>

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf jmpbuffer;
static int     globval;
int main(void)
{
    int            autoval;
    register int   regival;
    volatile int   volaval;
```

```
    static int      statval;

    globval = 1; autoval = 2; regival = 3;
    volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }
    /* Change variables after setjmp, but before longjmp.*/
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* never returns */
    exit(0);
}

static void f1(int i, int j, int k, int l)
{
   printf("in f1():\n");
   printf("globval = %d, autoval = %d, regival = %d,"
      " volaval = %d, statval = %d\n", globval, i, j, k, l);
   f2();
}

static void f2(void)
{
   longjmp(jmpbuffer, 1);
}
```

1. If we compile and test the above program with and without compiler optimizations, the results are different:

```
$ cc testjmp.c              compile without any optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99

$ cc -O testjmp.c           compile with full optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

- The setjmp(3) manual page on one system states that variables stored in memory will have values as of the time of the longjmp, whereas variables in the CPU and floating-point registers are restored to their values when setjmp was called.
- This is indeed what we see when we run the program above. Without optimization, all five variables are stored in memory (the register hint is ignored for regival). When we enable

optimization, both autoval and regival go into registers, even though the former wasn't declared register, and the volatile variable stays in memory.

- The thing to realize with this example is that you must use the volatile attribute if you're writing portable code that uses nonlocal jumps. Anything else can change from one system to the next

7.11. 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

- The resource limits for a process are normally established by process 0 when the system is initialized and then inherited by each successive process. Each implementation has its own way of tuning the various limits.
- 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) A process can change its soft limit to a value less than or equal to its hard limit.
b) 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.
c) Only a superuser process can raise a hard limit.

- An infinite limit is specified by the constant RLIM_INFINITY.

The resource argument takes on one of the following values.

RLIMIT_AS	The maximum size in bytes of a process's total available memory. This affects the sbrk function and the mmap function.
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.
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.

Example

The program below prints out the current soft limit and hard limit for all the resource limits supported on the system.

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

#if defined(BSD) || defined(MACOS)
#include <sys/time.h>
#define FMT "%10lld "
#else
#define FMT "%10ld "
#endif

#define doit(name) pr_limits(#name, name)

static void pr_limits(char *, int);

int main(void)
{

#ifdef  RLIMIT_AS
   doit(RLIMIT_AS);
#endif
   doit(RLIMIT_CORE);
   doit(RLIMIT_CPU);
   doit(RLIMIT_DATA);
```

```
    doit(RLIMIT_FSIZE);
#ifdef  RLIMIT_LOCKS
    doit(RLIMIT_LOCKS);
#endif
#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
    doit(RLIMIT_NOFILE);
#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
#ifdef  RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
#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)
        printf("getrlimit error for %s", name);
    printf("%-14s ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite) ");
    else
        printf(FMT, limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)");
    else
        printf(FMT, limit.rlim_max);
    putchar((int)'\n');
}
```

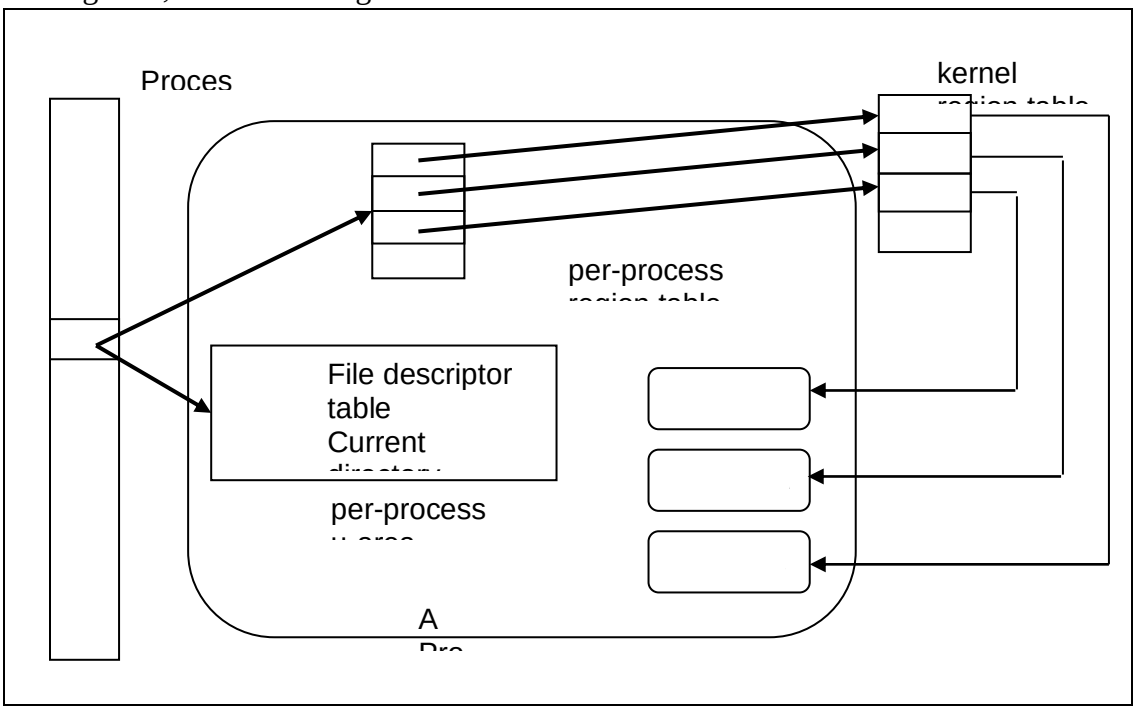
Running this program under FreeBSD gives us the following:

```
$ ./a.out
RLIMIT_CORE      (infinite) (infinite)
RLIMIT_CPU       (infinite) (infinite)
RLIMIT_DATA       536870912  536870912
RLIMIT_FSIZE     (infinite) (infinite)
RLIMIT_MEMLOCK   (infinite) (infinite)
RLIMIT_NOFILE         1735       1735
RLIMIT_NPROC           867        867
RLIMIT_RSS       (infinite) (infinite)
RLIMIT_SBSIZE    (infinite) (infinite)
```

RLIMIT_STACK 67108864 67108864
RLIMIT_VMEM (infinite) (infinite)

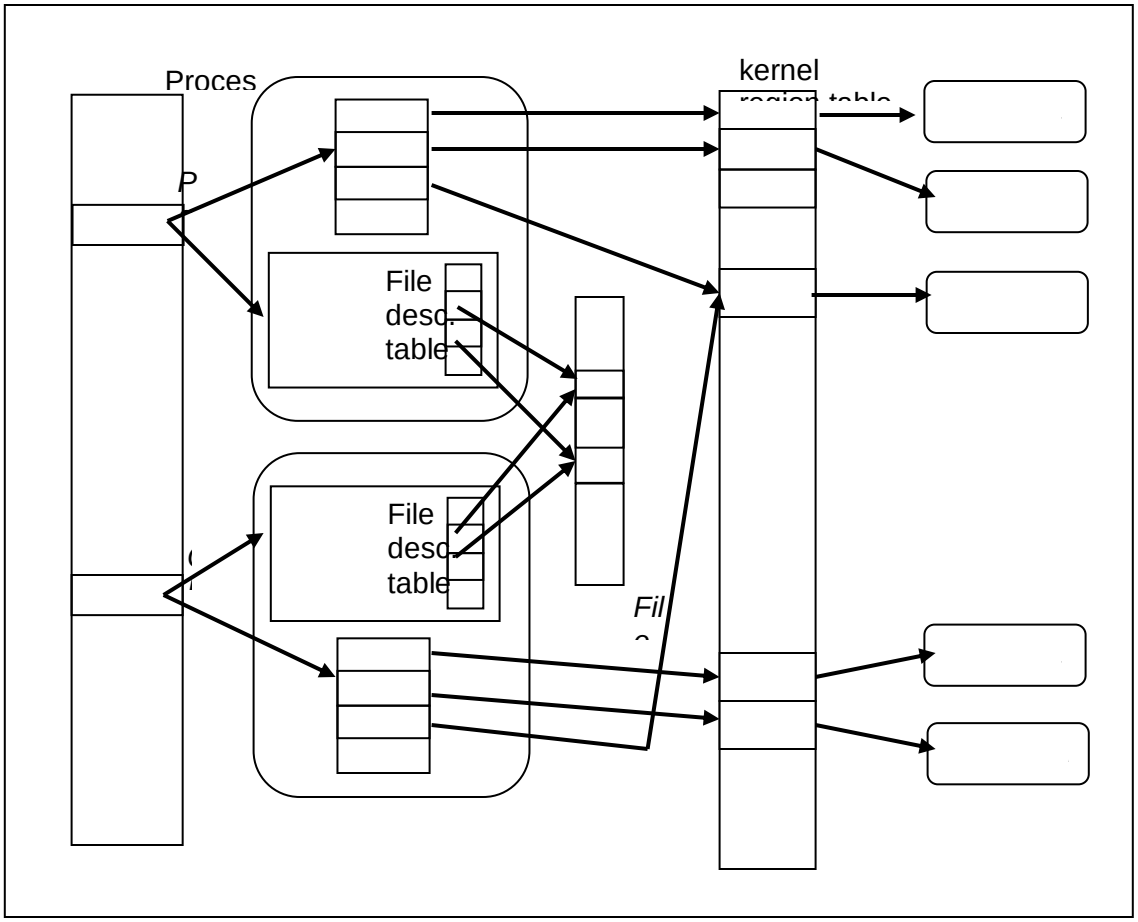
7.12 UNIX Kernel Support for processes.

- The data structure and execution of processes are dependent on operating system implementation.
- As shown in the figure below, a UNIX process consists minimally of a text segment, 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 contains the program text of a process in machine—executable instruction code format. A data segment contains static and global variables and their corresponding data. A stack segment contains a run-time stack. A stack provides storage for function arguments, automatic variables, and return address of all active functions for a process at any time.
- A UNIX kernel has a process table that keeps track of all active processes. Some of the processes belong to the kernel, they are called *system processes*. The majority of processes are associated with the users who are logged in.
- Each entry in the process table contains pointers to the text, data, stack segments and the U-area of a process.
- The U-area is an extension of a process table entry and contains other process specific data, such as the file descriptor table, current root, and working directory inode numbers, and a set of system –imposed process resource limits, etc.
- All processes in UNIX system, except the first process (process 0) which is created by the system boot code, are created via the *fork* system call.
- After the fork system call both the parent and child processes resume execution at the return of the fork function.

- As shown in the figure below, when a process is created by fork, it contains duplicate copies of the text, data, and stack segments of its parent. Also, it has an FDT that contains references to the same opened files as its parent, such that they both share the same file pointer to each opened file.



Furthermore, the process is assigned the following attributes which are either inherited by from its parent or set by the kernel.

a) ***A real user identification number (rUID):*** the user ID of a user who created the parent process.
b) ***A real group identification number (rGID)*****:** the group ID of a user who created the parent process.
c) ***An effective user identification number (eUID)*****:** this is normally the same as the real UID, except when the file that was executed to create the process has its set UID flag turned on, in that case the eUID will take on the UID of the file.
d) ***An effective group identification number (eGID)*****:** this is normally the same as the real GID, except when the file that was executed to create the process has its set UID flag turned on, in that case the eGID will take on the GID of the file.

e) ***Saved set-UID and saved set-GID*:** these are the assigned eUID and eGID, respectively of the process.
f) ***Process group identification number (PGID) and session identification number (SID)*:** these identify the process group and session of which the process is member.
g) ***Supplementary group identification numbers*:** this is a set of additional group IDs for a user who created the process.
h) ***Current Directory*:** this is the reference (inode number) to a working directory file.
i) ***Root Directory*:** this is the reference (inode number) to a root directory file.
j) ***Signal handling*:** the signal handling settings.
k) ***Signal mask*:** a signal mask that specifies which signals are to be blocked.
l) ***Umask*:** a file mode mask that is used in creation of files to specify which accession rights should be taken out.
m) ***Nice value*:** the process scheduling priority value.
n) ***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.
    a) ***Process identification number (PID)***: an integer identification number that is unique per process in an entire operating system.
    b) ***Parent process identification number (PPID)*:** the parent process ID.
    c) ***Pending signals*:** the set of signals that are pending delivery to the parent process. This is reset to none in the child process.
    d) ***Alarm clock time*:** the process alarm clock time is reset to zero in the child process.
    e) ***File locks*:** the set of file locks owned by the parent process is not inherited by the child process.
- After *fork* a parent process may choose to suspend its execution until its child process terminates by calling the *wait* or *waitpid* system call, or it may continue execution independently of its child process. In the later case, the parent process may use the *signal* or *sigaction* function to detect or ignore the child process termination.
- A process terminates its execution by calling the *_exit* system call. The argument to the *_exit* call is the exit status code of the process. By convention, an exit status code of zero means the process has completed its execution successfully, and any non zero exit code indicates failure has occurred.
- A process can execute a different program by calling the *exec* system call. If the call succeeds, the kernel will replace the process's existing text, data and stack segments with a new set that represents the new program to be executed.
- However, the process is still the same process (the PID and PPID are the same), and its FDT opened directory stream remain the same (except that those FDs which have their *close-on-exec* flag set via *fcntl* system call will be closed upon *exec'* ing).
- When the *exec'* ed program completes its execution, it terminates the process. The exit status code of the program may be polled by the process's parent via the *wait* or *waitpid* function.

- *fork and exec* are commonly used together to spawn a subprocess to execute a different program. For example UNIX shell executes each user command by calling *fork and exec* to execute the requested command in a child process.

The advantage of this method are:

- The 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.

- Two or more related processes (parent to child, or child to child with the same parent) may communicate with others by setting up unnamed pipes among them.
- For unrelated processes, they can communicate using named pipe or interprocess communication methods.

Chapter 8 PROCESS CONTROL

8.1. Introduction

Process control includes the creation of new processes, program execution, and process termination. We will also look at the various IDs that are the property of the process—real, effective, and saved; user and group IDs—and how they're affected by the process control primitives. Interpreter files and the system function and the process accounting provided by most UNIX systems.

8.2. Process Identifiers

Every process has a unique process ID, a non-negative integer. Although unique, processes IDs are reused. As processes terminate, their IDs become candidates for reuse. Most UNIX systems implement algorithms to delay reuse, however, so that newly created processes are assigned IDs different from those used by processes that terminated recently. This prevents a new process from being mistaken for the previous process to have used the same ID.

1. Process ID 0 is usually the scheduler process and is often known as the swapper. No program on disk corresponds to this process, which is part of the kernel and is known as a system process.

2. Process ID 1 is usually the init process and is invoked by the kernel at the end of the bootstrap procedure. The program file for this process was /etc/init in older versions of the UNIX System and is /sbin/init in newer versions. This process is responsible for bringing up a UNIX system after the kernel has been bootstrapped.

init usually reads the system-dependent initialization files—the /etc/rc* files or /etc/inittab and the files in /etc/init.d—and brings the system to a certain state, such as multi-user. The init process never dies. It is a normal user process, not a system process within the kernel, although it does run with *superuser* privileges.

Each UNIX System implementation has its own set of kernel processes that provide operating system services. For example, on some virtual memory implementations of the UNIX System, process ID 2 is the *page daemon*. This process is responsible for supporting the paging of the virtual memory system.

In addition to the process ID, there are other identifiers for every process. The following functions return these 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
```

Note that none of these functions has an error return.

8.3. fork Function

An existing process can create a new one by calling the fork function. The prototype for the fork function is:

```
#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.

For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory. The parent and the child share the text segment.

Current implementations don't perform a complete copy of the parent's data, stack, and heap, since a fork is often followed by an exec. Instead, a technique called *copy-on-write (COW)* is used. These regions are shared by the parent and the child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a "page" in a virtual memory system.

Example

The program below demonstrates the fork function, showing how changes to variables in a child process do not affect the value of the variables in the parent process.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
int     glob = 6;     /* external variable in initialized data */
char    buf[] = "a write to stdout\n";
int main(void)
{
   int      var;     /* automatic variable on the stack */
   pid_t    pid;
   var = 88;
   if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
      perror("write");
   printf("before fork\n");    /* we don't flush stdout */
   if ((pid = fork()) < 0) {
      perror("fork");
```

```
    } 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);
}
```

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

$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

We never know whether the child starts executing before the parent or vice versa. This depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize, some form of inter-process communication is required. In the program shown above, we simply have the parent put itself to sleep for 2 seconds, to let the child execute. The write function is not buffered. Because write is called before the fork, its data is written once to standard output. The standard I/O library, however, is buffered and that the standard output is line buffered if it's connected to a terminal device; otherwise, it's fully buffered.

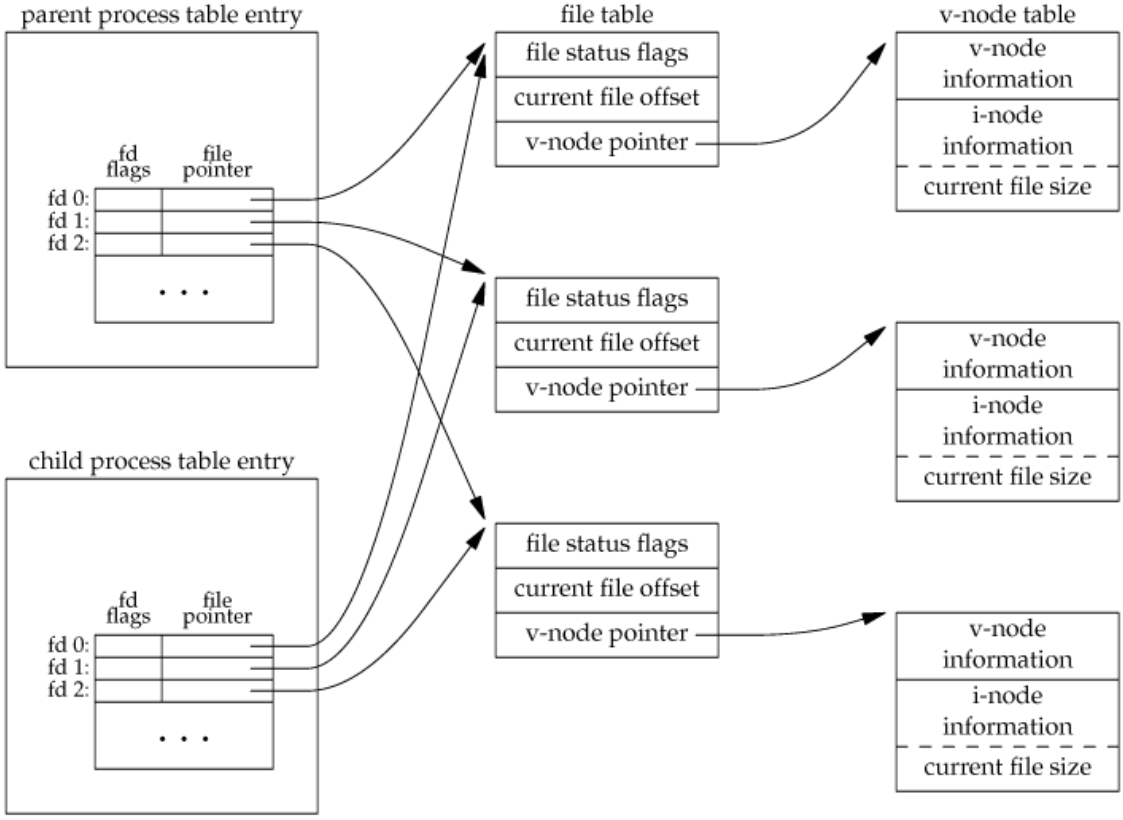
When we run the program interactively, we get only a single copy of the printf line, because the standard output buffer is flushed by the newline. But when we redirect standard output to a file, we get two copies of the printf line. In this second case, the printf before the fork is called once, but the line remains in the buffer when fork is called. This buffer is then copied into the child when the parent's data space is copied to the child. Both the parent and the child now have a standard I/O buffer with this line in it. The second printf, right before the exit, just appends its data to the existing buffer. When each process terminates, its copy of the buffer is finally flushed.

File Sharing

When we redirect the standard output of the parent from the above program, the child's standard output is also redirected. One characteristic of fork is that all file descriptors that are open in the parent are duplicated in the child. The parent and the child share a file table entry for every open descriptor.

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 figure below. It is important that the parent and the child share the same file offset. Consider a process that forks a child, and 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, perhaps) 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.



1. If both parent and child write to the same descriptor, without any form of synchronization, such as having the parent wait for the child, their output will be intermixed
2. There are two normal cases for handling the descriptors after a fork.
    a. **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 will have their file offsets updated accordingly.
    b. **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.

Besides the open files, there are numerous other properties of the parent that are inherited by the child:

a. Real user ID, real group ID, effective user ID, effective group ID
b. Supplementary group IDs
c. Process group ID
d. Session ID
e. Controlling terminal
f. The set-user-ID and set-group-ID flags

g. Current working directory
h. Root directory
i. File mode creation mask
j. Signal mask and dispositions
k. The close-on-exec flag for any open file descriptors
l. Environment
m. Attached shared memory segments
n. Memory mappings
o. Resource limits

The differences between the parent and child are:

a. The return value from fork
b. The process IDs are different
c. 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
d. The child's tms_utime, tms_stime, tms_cutime, and tms_cstime values are set to 0
e. File locks set by the parent are not inherited by the child
f. Pending alarms are cleared for the child
g. The set of pending signals for the child is set to the empty set

The two main reasons for fork to fail are:

(a) If too many processes are already in the system, which usually means that something else is wrong.

(b) If the total number of processes for this real user ID exceeds the system's limit. Recall that CHILD_MAX specifies the maximum number of simultaneous processes per real user ID.

There are two uses for fork:

1. When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. The parent waits for a service request from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
2. When a process wants to execute a different program. This is common for shells. In this case, the child does an exec right after it returns from the fork.

8.4. vfork Function

The function vfork has the same calling sequence and same return values as fork. But the semantics of the two functions differ. The vfork function originated with 2.9BSD. It is intended to create a new process when the purpose of the new process is to exec a new program. The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls exec (or exit) right after the vfork.

Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System.

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

lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)

Example

The program below is a modified version of the program from fork function. We've replaced the call to fork with vfork and removed the write to standard output. Also, we don't need to have the parent call sleep, as we're guaranteed that it is put to sleep by the kernel until the child calls either exec or exit.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.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");   /* we don't flush stdio */
   if ((pid = vfork()) < 0) {
      perror("vfork");
   } else if (pid == 0) {      /* child */
      glob++;                  /* modify parent's variables */
      var++;
      _exit(0);                /* child terminates */
   }
   /*
    * Parent continues here.
    */
   printf("pid=%d, glob=%d, var=%d\n", getpid(), glob, var);
   exit(0);
}
```

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

Here, the incrementing of the variables done by the child changes the values in the parent. Because the child runs in the address space of the parent. This behavior, however, differs from fork.

8.5. exit Functions

As we have seen a process can terminate normally in five ways:

a. Executing a return from the main function. This is equivalent to calling exit.

b. Calling the exit function. This function is defined by ISO C and includes the calling of all exit handlers that have been registered by calling atexit and closing all standard I/O streams. Because ISO C does not deal with file descriptors, multiple processes (parents and children), and job control, the definition of this function is incomplete for a UNIX system.

c. Calling the _exit or _Exit function. ISO C defines _Exit to provide a way for a process to terminate without running exit handlers or signal handlers. Whether or not standard I/O streams are flushed depends on the implementation. On UNIX systems, _Exit and _exit are synonymous and do not flush standard I/O streams. The _exit function is called by exit and handles the UNIX system-specific details; _exit is specified by POSIX.1.

d. Executing a return from the start routine of the last thread in the process. The return value of the thread is not used as the return value of the process, however. When the last thread returns from its start routine, the process exits with a termination status of 0.

e. Calling the pthread_exit function from the last thread in the process. As with the previous case, the exit status of the process in this situation is always 0, regardless of the argument passed to pthread_exit.

The three forms of abnormal termination are as follows:

a. Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.

b. When the process receives certain signals. The signal can be generated by the process itself—for example, by calling the abort function—by some other process, or by the kernel. Examples of signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.

c. The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later, the target thread terminates.

Regardless of how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and the like.

For any of the preceding cases, we want the terminating process to be able to notify its parent how it terminated. For the three exit functions (exit, _exit, and _Exit), this is done by passing an exit status as the argument to the function.
In the case of an abnormal termination, however, the kernel, not the process, generates a termination status to indicate the reason for the abnormal termination. In any case, the parent of the process can obtain the termination status from either the wait or the waitpid function.

Note that we differentiate between the exit status, which is the argument to one of the three exit functions or the return value from main, and the termination status. The exit status is converted into a termination status by the kernel when _exit is finally called.
The Table 5a below describes the various ways the parent can examine the termination status of a child. If the child terminated normally, the parent can obtain the exit status of the child.

Table 8a: Macros to examine the termination status returned by wait and waitpid

Macro	Description
WIFEXITED(status)	True if status was returned for a child that terminated normally. In this case, we can execute WEXITSTATUS (status) to fetch the low-order 8 bits of the argument that the child passed to exit, _exit, or _Exit.
WIFSIGNALED (status)	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute WTERMSIG (status) to fetch the signal number that caused the termination. Additionally, some implementations (but not the Single UNIX Specification) define the macro WCOREDUMP (status) that returns true if a core file of the terminated process was generated.
WIFSTOPPED (status)	True if status was returned for a child that is currently stopped. In this case, we can execute WSTOPSIG (status) to fetch the signal number that caused the child to stop.
WIFCONTINUED (status)	True if status was returned for a child that has been continued after a job control stop (XSI extension to POSIX.1; waitpid only).

But what happens if the parent terminates before the child?
The init process becomes the parent process of any process whose parent terminates.
Whenever a process terminates, the kernel goes through all active processes to see whether the terminating process is the parent of any process that still exists. If so, the parent process ID of the surviving process is changed to be 1 (the process ID of init). This way, we are guaranteed that every process has a parent.

Another condition we have to worry about is when a child terminates before its parent. If the child completely disappeared, the parent wouldn't be able to fetch its termination status when and if the parent were finally ready to check if the child had terminated. The kernel keeps a small amount of information for every terminating process, so that the information is available when the parent of the terminating process calls wait or waitpid. Minimally, this information consists of the process ID, the termination status of the process, and the amount of CPU time taken by the process.

The kernel can discard all the memory used by the process and close its open files. In UNIX System terminology, a process that has terminated, but whose parent has not yet waited for it, is called a ***zombie***.

The ps command prints the state of a zombie process as Z. If we write a long-running program that forks many child processes, they become zombies unless we wait for them and fetch their termination status.

The final condition to consider is this**: what happens when a process that has been inherited by init terminates? Does it become a zombie?**
The answer is "no," because init is written so that whenever one of its children terminates, init calls one of the wait functions to fetch the termination status. By doing this, init prevents the system from being clogged by zombies.

8.6. wait and waitpid Functions

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event—it can happen

at any time while the parent is running—this signal is the asynchronous notification from the kernel to the parent.

The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. The default action for this signal is to be ignored.

A process that calls wait or waitpid can

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

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

The prototypes of wait and waitpid are as:

```
#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, 0 (see later), 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. We can always tell which child terminated, because the process ID is returned by the function.

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 integer status that these two functions return has been defined by the implementation, with certain bits indicating the exit status (for a normal return), other bits indicating the signal number (for an abnormal return), one bit to indicate whether a core file was generated, and so on.

POSIX.1 specifies that the termination status is to be looked at using various macros that are defined in <sys/wait.h>. Four mutually exclusive macros tell us how the process terminated, and they all begin with WIF as shown in **Table 8a.**

Example

1. The function pr_exit defined below uses the macros from table 8a to print a description of the termination status. Note that this function handles the WCOREDUMP macro, if it is defined.

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

void pr_exit(int status)
{
   if (WIFEXITED(status))
      printf("normal termination, exit status = %d\n",
            WEXITSTATUS(status));
   else if (WIFSIGNALED(status))
      printf("abnormal termination, signal number = %d%s\n",
            WTERMSIG(status),
#ifdef  WCOREDUMP
           WCOREDUMP(status) ? " (core file generated)" : "");
#else
            "");
#endif
   else if (WIFSTOPPED(status))
      printf("child stopped, signal number = %d\n",
            WSTOPSIG(status));
}
```

2. The program shown below calls the pr_exit function, demonstrating the various values for the termination status.

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

int main(void)
{
   pid_t   pid;
   int     status;

   if ((pid = fork()) < 0)
      perror("fork");
   else if (pid == 0)            /* child */
      exit(7);

   if (wait(&status) != pid)       /* wait for child */
      perror("wait");
   pr_exit(status);              /* and print its status */
```

```
    if ((pid = fork()) < 0)
        perror("fork");
    else if (pid == 0)              /* child */
        abort();                    /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        perror("wait");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        perror("fork");
    else if (pid == 0)              /* child */
        status /= 0;                /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        perror("wait");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

3. What if we want to wait for a specific process to terminate (assuming we know which process ID we want to wait for)?

- In older versions of the UNIX System, we would have to call wait and compare the returned process ID with the one we're interested in. If the terminated process wasn't the one, we wanted, we would have to save the process ID and termination status and call wait again.
- We would need to continue doing this until the desired process terminated. The next time we wanted to wait for a specific process, we would go through the list of already terminated processes to see whether we had already waited for it, and if not, call wait again.

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

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

The waitpid function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed to by statloc.

With wait, the only real error is if the calling process has no children.

With waitpid, however, it's also possible to get an error if the specified process or process group does not exist or is not a child of the calling process.

The options argument lets us further control the operation of waitpid. This argument is either 0 or is constructed from the bitwise OR of the constants in table 8b.

Table 8b: The options constants for waitpid

Constant	Description
WCONTINUED	If the implementation supports job control, the status of any child specified by pid that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI extension to POSIX.1).
WNOHANG	The waitpid function will not block if a child specified by pid is not immediately available. In this case, the return value is 0.
WUNTRACED	If the implementation supports job control, the status of any child specified by pid that has stopped, and whose status has not been reported since it has stopped, is returned. The WIFSTOPPED macro determines whether the return value corresponds to a stopped child process.

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

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

b. 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.

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

Example

If we want to write a process so that it forks a child but we don't want to wait for the child to complete and we don't want the child to become a zombie until we terminate, the trick is to call fork twice.

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

int main(void)
{
   pid_t   pid;

   if ((pid = fork()) < 0) {
      perror("fork error");
   } else if (pid == 0) {     /* first child */
      if ((pid = fork()) < 0)
         perror("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 */
        perror("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);
}
```

We call sleep in the second child to ensure that the first child terminates before printing the parent process ID. After a fork, either the parent or the child can continue executing; we never know which will resume execution first. If we didn't put the second child to sleep, and if it resumed execution after the fork before its parent, the parent process ID that it printed would be that of its parent, not process ID 1.

Executing the program above gives us

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

8.7. waitid Function

The waitid function is similar to waitpid, but provides extra flexibility.

The prototype of the function is as

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

Returns: 0 if OK, 1 on error

Like waitpid, waitid allows a process to specify which children to wait for. Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used. The id parameter is interpreted based on the value of idtype. The types supported are summarized in table 8c.

Table 8c. The idtype constants for waitid

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

The options argument is a bitwise OR of the flags shown in table 8d. These flags indicate which state changes the caller is interested in.

Table 8d. The options constants for waitid

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.

The infop argument is a pointer to a siginfo structure. This structure contains detailed information about the signal generated that caused the state change in the child process.

8.8. wait3 and wait4 Functions

Most UNIX system implementations provide two additional functions: wait3 and wait4. The only feature provided by these two functions that is not provided by the wait, waitid, and waitpid functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

The prototype 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, 0, or 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, and the like.

8.9. Race Conditions

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

We see a potential race condition in the program below when the second child printed its parent process ID.

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

int main(void)
{
   pid_t   pid;

   if ((pid = fork()) < 0) {
      perror("fork error");
   } else if (pid == 0) {     /* first child */
      if ((pid = fork()) < 0)
         perror("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 */
      perror("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);
}
```

If the second child runs before the first child, then its parent process will be the first child. But if the first child runs first and has enough time to exit, then the parent process of the second child is init. Even calling sleep, as we did, guarantees nothing. If the system was heavily loaded, the second child could resume after sleep returns, before the first child has a chance to run.

Problems of this form can be difficult to debug because they tend to work "most of the time." A process that wants to wait for a child to terminate must call one of the wait functions.

If a process wants to wait for its parent to terminate, as in the program above, a loop of the following form could be used:

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

The problem with this type of loop, ***called polling***, is that it wastes CPU time, as the caller is awakened every second to test the condition. To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes. Signals can be used, and various forms of interprocess communication (IPC) can also be used.

For a parent and child relationship, we often have the following scenario.

After the fork, both the parent and the child have something to do. For example, the parent could update a record in a log file with the child's process ID, and the child might have to create a file for the parent.
In this example, we require that each process tell the other when it has finished its initial set of operations, and that each wait for the other to complete, before heading off on its own. The following code illustrates this scenario:

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

TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ((pid = fork()) < 0) {
   perror("fork");
} else if (pid == 0) {          /* child */

   /* child does whatever is necessary ... */

   TELL_PARENT(getppid());     /* tell parent we're done */
   WAIT_PARENT();              /* and wait for parent */

   /* and the child continues on its way ... */

   exit(0);
}

/* parent does whatever is necessary ... */
TELL_CHILD(pid);            /* tell child we're done */
WAIT_CHILD();               /* and wait for child */

/* and the parent continues on its way ... */

exit(0);
```

The five routines TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT, and WAIT_CHILD can be either macros or functions. The ways to implement these TELL and WAIT routines may be any forms of IPC or signals. Let's look at an example that uses these five routines.

Example

The program below outputs two strings: one from the child and one from the parent.

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

static void charatatime(char *);

int main(void)
{
   pid_t   pid;
```

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

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.

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

We need to change the program above to use the TELL and WAIT functions.

The program below does this. The lines preceded by a plus sign are new lines.

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

  static void charatatime(char *);

  int main(void)
  {
      pid_t   pid;

+     TELL_WAIT();

      if ((pid = fork()) < 0) {
          perror("fork error");
      } else if (pid == 0) {
+         WAIT_PARENT();      /* parent goes first */
          charatatime("output from child\n");
      } else {
```

```
        charatatime("output from parent\n");
+       TELL_CHILD(pid);
    }
    exit(0);
  }
  static void charatatime(char *str)
  {
    char    *ptr;
    int     c;

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

In the program shown above, the parent goes first. The child goes first if we change the lines following the fork to be

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

8.10. exec Functions

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

There are six different exec functions
These six functions round out the UNIX System process control primitives. With fork, we can create new processes; and with the exec functions, we can initiate new programs.
The prototypes of the exec functions are as:

```
#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
```

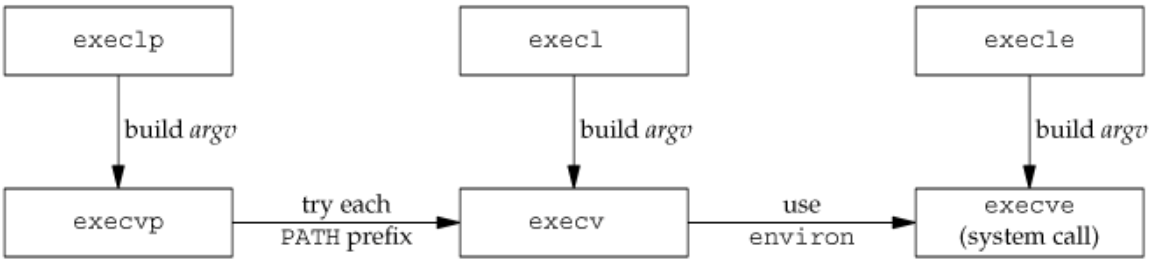
The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument.
When a filename argument is specified

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

- Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.

The PATH variable contains a list of directories, called path prefixes that are separated by colons. For example, the **name=value** environment string

PATH=/bin:/usr/bin:/usr/local/bin/:.



specifies four directories to search. The last path prefix specifies the current directory.

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

Before using ISO C prototypes, the normal way to show the command-line arguments for the three functions execl, execle, and execlp was

char *arg0, char *arg1, ..., char *argn, (char *)0

The final difference is the passing of the environment list to the new program. The two functions whose names end in an **e** (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings.

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

In many UNIX system implementations, only one of these six functions, execve, is a system call within the kernel. The other five are just library functions that eventually invoke this system call. We can illustrate the relationship among these six functions as shown in figure below.

In this arrangement, the library functions execlp and execvp process the PATH environment variable, looking for the first path prefix that contains an executable file named filename.

Example

The program in below demonstrates the exec functions.

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

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

```
int main(void)
{
   pid_t   pid;

   if ((pid = fork()) < 0) {
      perror("fork error");
   } else if (pid == 0) {  /* specify pathname, specify environment */
      if (execle("/home/sar/bin/echoall", "echoall", "myarg1",  "MY ARG2", (char *)0, env_init) <
0)
         perror("execle error");
   }

   if (waitpid(pid, NULL, 0) < 0)
      perror("wait error");

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

We first call execle, which requires a pathname and a specific environment. The next call is to execlp, which uses a filename and passes the caller's environment to the new program. The program echoall that is executed twice in the above program and is listed below. It is a trivial program that echoes all its command-line arguments and its entire environment list.

```
#include <stdio.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);
}
```

When we execute the program, we get

```
$ ./a.out
  argv[0]: echoall
```

```
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
```