UNIX Programming (18CS56)

Module 3

(I) Introduction to UNIX File APIs

- **API** stands for Application Programming Interface.
- An API is a computing interface that allows interactions between two applications.
- UNIX systems contain a set of functions that can be called by users' programs to perform system-specific functions.
- This set of functions defines UNIX APIs.
- UNIX File APIs are APIs that can be used to perform specific functions on the files.
- There are several functions that can be performed on files using these APIs.
- Some of the functions on files are:
  - Create files
  - Open files
  - Transfer data to and from files
  - Close files
- Some of the functions on files are:
  - Remove files
  - Query file attributes
  - Change file attributes
  - Truncate files

(II) Introduction to General File APIs

- Different types of files in the UNIX file system are:
  - Regular file
  - Directory file
  - Device file
  - Character device file
  - Block device file
  - Symbolic link file
- UNIX consists of a special set of APIs that can create and manipulate these different types of files.

- The following table lists the different general file APIs that are used in UNIX.

API	Significance
open	Opens a file for data access
read	Reads data from a file
write	Writes data to a file
lseek	Allows random access of data in a file
close	Terminates the connection to a file
stat, fstat	Queries attributes of a file
chmod	Changes access permission of a file
chown	Changes UID or GID of a file
utime	Changes last modification and access timestamps of a file
link	Creates a hard link to a file
unlink	Deletes a hard link to a file
umask	Sets default file creation mask

(III) open File API

- **open** function establishes a connection between a process and a file.
- It can be used to create brand new files.
- Any process can call **open** function to get a file descriptor to refer to the created file.
- The file descriptor is used in the **read** and **write** system calls to access the file content.
- The prototype of the *open* File API is:

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

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

- *path_name* is the path name of the file.
- It can be absolute pathname or relative pathname.
- If it is a symbolic link, the function will resolve the link reference to a file to which the link refers.
- *access_mode* is an integer value that specifies how the file is to be accessed by the calling process.
- Value of *access_mode* should be one of the manifested constants defined in the <*fcntl.h*> header.

- The following table lists the different access mode flags.

Access Mode Flag	Significance
O_RDONLY	Opens the file for read-only
O_RDONLY	Opens the file for write-only
O_RDONLY	Opens the file for read and write

- There are six modifier flags that can be used to alter the access mechanism of a file.
- Bitwise-OR operation can be performed with the previously mentioned *access_mode* flags to alter the access mechanism.
- The following table lists the different access modifier flags.

Access Modifier Flag	Significance
O_APPEND	Appends data to the end of the file
O_CREAT	Creates the file if it does not exist
O_EXCL	Used with **O_CREAT** flag. Causes *open* to fail if the named file already exists.
O_TRUNC	If the file exists, it discards the file contents and sets the file size to 0 bytes.
O_NONBLOCK	Subsequent read or write on the file should be non-blocking.
O_NOCTTY	Not to use the named terminal device file as calling process control terminal

- An example statement using *open* File API is:

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

- File */usr/xyz/textbook* is opened for read and write operation in append mode.
- If a file has to be opened for read-only, file should already exist and no other modifier flags can be used.
- If a file is opened for write-only or read-write, any modifier flags can be specified.
- **O_APPEND**, **O_TRUNC**, **O_CREAT** and **O_EXCL** are applicable to regular files only.
- **O_NONBLOCK** is applicable to FIFO and device files only.
- **O_NOCTTY** is applicable to terminal device files only.
- **O_APPEND** flag – data written to a named file will be appended at the end of the file.
  If not specified, data can be written anywhere in the file.
- **O_TRUNC** flag – if a named file already exists, the *open* function discards its content.
  If not specified, data will not be altered by the *open* function.

- **O_CREAT** flag – if a named file does not exist, the *open* function should create it. If a named file does exist, then the *open* function has no effect on it.
- **O_NONBLOCK** flag – if the *open* and any subsequent *read* or *write* function calls on a named file will block a calling process, kernel should abort the function immediately and return to the process.
- **O_NOCTTY** flag – defined in POSIX.1
  If a process has no controlling terminal and it opens a terminal device file, that terminal will not be the controlling terminal of the process.
- *permission* argument is required only if **O_CREAT** flag is set in the *access_mode* argument.
- It specifies the access permission of the file for its owner, group member and all other people.
- POSIX.1 defines the *permission* data type as *mode_t*, and its value should be constructed based on the manifested constants defined in the <*sys/stat.h*> header.
- These constants are aliases to the octal integer values used in UNIX System V.
- *permission* value is modified by its calling process *umask* value.
- This value specifies some access rights to be masked off (taken away) automatically on any files created by the process.
- A process umask is inherited from its parent process, and its value can be queried or changed by the *umask* system call.
- The prototype of the *umask* API is:

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

- *umask* function takes a new umask value as an argument.
- This value will be used by the calling process from then onwards, and the function returns the old umask value.
- *Example* – assign current umask value to the variable *old_mask*, and sets new umask value to "no execute for group" and "no write-execute for others."

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

- *open* function takes its *permission* argument value and bitwise-ANDs it with the one's complement of the calling process umask value.
- The final access permission to be assigned to any new file that is created is given by:

```
actual_permission = permission & ~umask_value
```

- Example usage of *umask* to alter permissions –

```
actual_permission = 0557 & (~031) = 0546
```

- Return value of *open* function is –1 if it fails and *errno* contains error status value.
- If the API succeeds, the return value is a file descriptor that can be used to reference the file in other system calls.
- Value of file descriptor lies between 0 and OPEN_MAX-1.

(IV) creat File API

- **creat** system call is used to create new regular files.
- The prototype of the **creat** File API is:

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

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

- *creat* has become obsolete because *open* API now has **O_CREAT** flag, which is used to create and open regular files.
- *creat* can be implemented using the open function as:

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

(V) read File API

- *read* function – fetches a fixed size block of data from a file referenced by a given file descriptor.
- The prototype of the *read* file API is:

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

ssize_t read(int fdesc, void* buf, size_t size);
```

- *fdesc* – integer file descriptor that refers to an opened file.
- *buf* – address of a buffer holding any data read.
- *size* – specifies how many bytes of data are to be read from the file.
- *size_t* data type is defined in the <sys/types.h> header.
- It should be the same as *unsigned int*.
- *read* File API can read text or binary files.
- This is the reason data type of *buf* is a universal pointer (*void**).
- Return value of *read* function is the number of bytes successfully read and stored in the *buf* argument.
- It should normally be equal to the *size* value.
- If a file contains less than *size* bytes of data remaining to be read, return value of *read* function will be less than that of *size*.
- If end of file is reached, *read* will return zero.
- ssize_t is usually defined as *int* in the <sys/types.h> header.
- Users should not set *size* to exceed INT_MAX in any *read* function call.
- This ensures that the function return value can reflect the actual number of bytes read.

- Example code fragment to read one or more records sequentially of *struct sample*-typed data from a file called *dbase*:

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

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

while(read(fd, &varX, sizeof(varX)) > 0)
        /*process data stored in varX*/
```

- *Case* – interruption of *read* function call by a caught signal and OS does not restart the system call automatically.
- Two possible behaviours of the *read* function are allowed by POSIX.1 specification.
- First behaviour – *read* function will return a value of -1, *errno* will be set to EINTR and all the data read in the call will be discarded.
  Process cannot recover the data in this case.
- Second behaviour – *read* function will return the number of bytes of data read prior to the signal interruption.
  Process can continue to read the file.
- BSD UNIX – the kernel automatically restarts any system call after a signal interruption.
- Return value of *read* will be same as that in a normal execution.
- UNIX System V.4 – user can specify whether the kernel will restart any system call that is interrupted by a signal.
- Behaviour of *read* function maybe similar to that of BSD UNIX for restartable signals, or to that of UNIX System V.3 or POSIX.1 FIPS systems for non-restartable signals.
- *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 then specify the **O_NONBLOCK** and **O_NDELAY** flags on a file descriptor to request non-blocking read operations on the corresponding file.

(VI) write File API

- *write* function – puts a fixed block of data to a file referenced by a given file descriptor.
- The prototype of the *write* File API is:

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

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

- *fdesc* – integer file descriptor that refers to an opened file.
- *buf* – address of a buffer which contains any data to be written to the file.
- *size* – specifies how many bytes of data are in the *buf* argument.
- Like the *read* API, *write* API can perform write operations on text or binary files.
- This is the reason data type of *buf* is a universal pointer (*void**).
- Return value of *write* function is the number of bytes successfully written to a file.
- It should normally be equal to the *size* value.
- If a write operation causes file size to exceed a system-imposed limit or if the disk is full, the return value of write will be the actual number of bytes written before the function was aborted.
- Example code fragment to write ten records sequentially of *struct sample*-typed data to a file called *dbase2*:

```
struct sample{
        int x;
        double y;
        char* z;
}varX[10];

int fd = open("dbase2", O_WRONLY);

/*initialize varX array here... */

write(fd, (void*)varX, sizeof(varX));
```

- *Case* – interruption of *write* function call by a caught signal and OS does not restart the system call automatically.
- Two possible behaviours of the *write* function are allowed by POSIX.1 specification.
- *First behaviour* – *write* function will return a value of -1, *errno* will be set to EINTR and all the data written in the call will be discarded.
  Process cannot recover the data in this case.

- *Second behaviour* – *write* function will return the number of bytes of data written prior to the signal interruption.
  Process can continue to write the file.
- BSD UNIX – the kernel automatically restarts any system call after a signal interruption.
- Return value of *write* will be same as that in a normal execution.
- UNIX System V.4 – user can specify whether the kernel will restart any system call that is interrupted by a signal.
- Behaviour of *write* function maybe similar to that of BSD UNIX for restartable signals, or to that of UNIX System V.3 or POSIX.1 FIPS systems for non-restartable signals.
- *write* function may block a calling process execution if it is writing to a FIFO or device file and data is not yet available to satisfy the write request.
- Users may then specify the **O_NONBLOCK** and **O_NDELAY** flags on a file descriptor to request non-blocking write operations on the corresponding file.

(VII) close File API

- *close* function – disconnects a file from a process.
- The prototype of the *close* File API is:

```
#include<unistd.h>

int close(int fdesc);
```

- *fdesc* – integer file descriptor that refers to an opened file.
- Return value of *close* is zero if the call succeeds, or -1 if it fails (*errno* will contain the respective error code).
- *close* function frees unused file descriptors so that they can be used to reference other files.
- A process may open upto OPEN_MAX files at any one time.
- *close* function allows a process to reuse file descriptors to access more than OPEN_MAX files during its execution.
- *close* function will deallocate system resources that support the operation of file descriptors, thereby reducing memory requirement of a process.
- If a process terminates without closing all the files it has opened, the kernel will close those files for the process.
- *iostream* class defines a *close* member function to close a file associated with an iostream object.

- This member function may be implemented using the *close* API as:

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

int iostream::close(){
        return close(this->fileno());
}
```

(VIII) fcntl File API

- *fcntl* function helps a user to query or set access control flags and the *close-on-exec* flag of any file descriptor.
- It can also be used to assign multiple file descriptors to reference the same file.
- The prototype of the *fcntl* File API is:

```
#include<fcntl.h>

int fcntl(int fdesc, int cmd, ...);
```

- *fdesc* – integer file descriptor that refers to an opened file.
- *cmd* – specifies which operations to perform on a file referenced by *fdesc* argument.
- Third argument is dependent on actual *cmd* value.
- The *cmd* values are given in the following table.

cmd value	Significance
F_GETFL	Return the access control flags of a file descriptor
F_SETFL	Sets or clears access control flags that are specified in third argument to *fcntl*. 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 return value is zero, flag is OFF. If return value is non-zero, flag is ON. 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*. Third argument to *fcntl* is an integer value. This value is 0 to clear the flag. This value is 1 to set the flag.
F_DUPFD	Duplicates the file descriptor *fdesc* with another file descriptor. Third argument to *fcntl* is an integer value, which specifies that the duplicated file descriptor must be ≥ that value. Return value of *fcntl* is the duplicated file descriptor.

- *fcntl* function is useful in changing the access control flag of a file descriptor.
- Example – after a file is opened for blocking read-write access and process needs to change the access to non-blocking and in write-append mode, it can call *fcntl* on the file descriptor.

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

- *close-on-exec* flag – if the process that owns the descriptor calls the *exec* API to execute a different program, the file descriptor should be closed by the kernel before the new program runs.
- Example – report the *close-on-exec* flag of a file descriptor *fdesc* and set it to ON afterward.

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

- *fcntl* function can be used to duplicate a file descriptor *fdesc* with another file descriptor.
- Results are two file descriptors referencing the same file with the same access mode (read / write, blocking / nonblocking access) and sharing the same file pointer to read / write the file.
- This is particularly useful in redirection of standard input / output to a reference file.
- Example – change the standard input of a process to a file called FOO

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

- *dup* and *dup2* functions perform same file duplication function as *fcntl*.

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

- *dup* function duplicates a file descriptor *fdesc* with the lowest unused file descriptor of a calling process.
- *dup* function duplicates a file descriptor *fdesc* using a *fd2* file descriptor, regardless of whether *fd2* is used to reference another file.
- Return value of *fcntl* is dependent on *cmd* value, but it is -1 if the function fails.

(IX) lseek File API

- *read* and *write* system calls are always relative to the current offset within a file.
- *lseek* can be used to change the file offset to a different value.
- *lseek* allows a process to perform random access of data on any opened file.
- It is incompatible with FIFO files, character device files and symbolic link files.
- The prototype of the *lseek* file API is:

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

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

- *fdesc* – 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 offset value.
- The reference location is specified by the *whence* argument.

whence value	Significance
SEEK_CUR	Current file pointer address
SEEK_SET	The beginning of a file
SEEK_END	The end of a file

- One cannot specify a negative *pos* value with the *whence* value set to SEEK_SET, as this will cause the function to assign a negative file offset.
- If *lseek* call will result in a new file offset that is beyond the current end-of-file, two outcomes are possible.
  - if a file is opened for read-only, *lseek* will fail.
  - 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.
- Data between the end-of-file and the new file offset address will be initialized with NULL characters.
- Return value of *lseek* is the new file offset address where the next read or write operation will occur, or -1 if the *lseek* call fails.
- *iostream* class defines the *tellg* and *seekg* functions to allows users to do random data access of any *iostream* object.

- *iostream:tellg* – calls *lseek* to return current file pointer associated with *iostream*

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

streampos iostream::tellg(){
        return (streampos)lseek(this->fileno(), (off_t)0, SEEK_CUR);
}
```

- *iostream:seekg* – calls *lseek* to alter file pointer associated with *iostream* object.

```
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;
}
```

(X) link File API

- *link* function – creates a new link for an existing file.
- Does not create a new file, instead it creates a new path for an existing file.
- The prototype of the *link* File API is:

```
#include<unistd.h>

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

- *cur_link* – path name of an existing file.
- *new_link* – 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* File API cannot be used to create hard links across file systems.
- *link* cannot be used on directory files unless it is called by a process that has superuser privileges.

- *ln* command in UNIX is implemented using the *link* File API.

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>

int main(int argc, char* argv[]){
        if(argc != 3){
                cerr << "usage:" << argv[0] << "<src_file> <dest_file>\n";
                return 0;
        }
        if(link(argv[1], argv[2]) == -1){
                perror("link");
                return 1;
        }
        return 0;
}
```

(XI) unlink File API

- *unlink* function deletes the 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.
- A file is removed from the file system when its hard link count is zero and no process has any file descriptor referencing that file.
- The prototype of the *unlink* File API is:

```
#include<unistd.h>

int unlink(const char* cur_link);
```

- *cur_link* – path name that references an existing file.
- Return value is 0 if the call succeeds, or -1 if the call fails.
- *unlink* cannot be used to remove a directory file unless the calling process has the superuser privilege.
- Possible causes of failure of *unlink* -
  - *cur_link* is invalid (no file exists with that name).
  - the calling process lacks access permission to remove that path name.
  - the *unlink* function is interrupted by a signal.
- ***remove*** function – similar to *unlink* function and is used to remove files or directories.
- The prototype of the *remove* File API is:

```
#include<stdio.h>

int remove(const char* path_name);
```

- It is similar to *unlink* for files, and to that of *rmdir* for directories.
- ***rename*** function – can be used to rename files or directories.
- The prototype of the *rename* File API is:

```
#include<stdio.h>

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

- Both *link* and *rename* will fail if new link to be created is in a different file system/partition.
- *mv* command in UNIX is implemented using the *link* and *unlink* File APIs.

```
#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;
}
```

- The program takes two command line arguments – *old_link* and *new_link*.
- It first checks that *old_link* and *new_link* are different path names.
  If they are the same, the program exits immediately, as there is nothing to change.
  If they are not the same, the program calls *link* to set up *new_link* as a new reference to *old_link*.
- If *link* fails, the program will return -1 as the error status.
  Otherwise, it will call *unlink* to remove the *old_link*, and its return value is that of the *unlink* call.

(XII) stat, fstat, lstat File APIs

- *stat* and *fstat* functions are used to retrieve the file attributes of a given file.
- Difference between *stat* and *fstat* –
  - *stat* – first argument is a file path name.
  - *fstat* – first argument is a file descriptor.
- The prototypes of the *stat* and *fstat* File APIs are as follows:

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

int stat(const char* path_name, struct stat* statv);
int fstat(const int fdesc, struct stat* statv);
```

- The first argument is a path name (*stat*) or a file descriptor (*fstat*).

- The second argument is the address of a *struct stat*-typed variable.
- *struct stat* data type is defined in the <sys/stat.h> header.
- Declaration of *struct stat* is given as follows:

```
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 ad 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 *stat* and *fstat* is 0 if they succeed or -1 if they fail, and *errno* gives the status code of error.
- Possible causes of failures for these APIs are –
  - file path or file descriptor is invalid.
  - calling process lacks permission to access the file.
  - functions interrupted by signal.
- *stat* and *fstat* cannot be used to obtain attributes of symbolic link files.
- Thus, *lstat* is used for link files.
- The prototype of the *lstat* function is:

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

- The behaviour of *lstat* is similar to that of *stat* for non-symbolic link files.
- If *path_name* argument to *lstat* is a symbolic link file, then *lstat* will return the attributes of the symbolic link file, and not the file it refers to.
- Implementation of *ls -l* can be done using *stat* API.
- The 7 attributes (10 fields) of the *ls -l* command are displayed.
  - File type and owner, group and others access rights
  - Hard link count of file
  - User name of file
  - Group name of file
  - File size in bytes (or major and minor device numbers if character/block device file)
  - Last modified timestamp
  - File name

```
#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";
#ifndef MAJOR
#define MINOR_BITS 8
#define MAJOR(dev) ((unsigned)dev >> MINOR_BITS)
#define MINOR(dev) (dev & MINOR_BITS)
#endif
```

```
static void display_file_type(ostream& ofs, int st_mode){
        switch(st_mode & S_IFMT){
                case S_IFDIR: ofs << 'd'; return;
                case S_IFCHR: ofs << 'c'; return;
                case S_IFBLK: ofs << 'b'; return;
                case S_IFREG: ofs << '-'; return;
                case S_IFLNK: ofs << 'l'; return;
                case S_IFIFO; ofs << 'p'; return;
        }
}
```

```
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] : '-';
        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 << '';
}
```

```
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;
        gr_p = getgrgid(statv.st_gid);
        pw_p = getpwuid(statv.st_uid);
        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;
        ofs << '' << ctime(&statv.st_mtime);
        ofs << '' << path_name << endl;
}
```

```
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;
}
```

- *st_mode* variable has several attributes – file type, owner/group/others access rights, *set-UID* and *set-GID* flags, and *sticky* bit.
- File types - **d** (directory file), **c** (character device file), **b** (block device file), **-** (regular file), **p** (FIFO file) and **l** (symbolic link file).
- Access permissions – **r** (read), **w** (write) and **x** (execute)
- *set-UID* of a file is ON - effective user ID of process created by executing that file will be same as file user ID.
- *set-GID* of a file is ON - effective group ID of process created by executing that file will be same as file group ID.
- Effective user ID and group ID of a process are used to determine access permission of process to any file.
- If there is a match between IDs, then the access is granted.
  If there is no match between IDs, then no permission can be altered.
- *set-UID* and *set-GID* are useful on processes that need superuser privileges.
- Effective user ID and group ID are used when process creates a file.
- *sticky* flag is set when the instruction code resides in swap memory even after process terminates, thereby making next execution start-up faster.
- It is reserved for frequently used programs and can be set/reset only by superuser.
- File size - files, directories and named pipes (major and minor numbers for device).

(XIII) access File API

- *access* function – checks the existence and/or access permission of user to a named file.
- The prototype of the *access* File API is:

```
#include<unistd.h>

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

- *path_name* – path name of an existing file.
- *flag* – contains one or more bit flags.
- *flag* argument value to an *access* call is composed by bitwise-ORing one or more bit flags.
  These bit flags are given in the table below.

Bit flag	Significance
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 read permission
X_OK	Checks whether a calling process has read permission

- The following example checks whether a user has read and write permissions on a file.

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

- If *flag* value is F_OK, the *access* function returns 0 if the *path_name* file exists and -1 otherwise.
- If *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 file group ID.
- It determines the appropriate category (owner, group or others) of access permissions in checking against the actual value of *flag*.
- *access* returns 0 if all the requested permission is permitted, and -1 otherwise.
- The following program uses *access* to determine whether a named file exists for each command line argument.
  If a named file does not exist, it will be created and initialized with a character string "Hello world."
  If a named file exists, the program will simply read data from the file.

```c
#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)){
                        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);
                /*for each command line argument*/
        }
}
```

(XIV) chmod, fchmod File APIs

- *chmod, fchmod* functions are used to change file access permissions for owner, group and others.
- They can also change *set-UID*, *set-GID* and *sticky* flags.
- To change the permissions, the calling process should have effective user ID of either the superuser or the owner of the file.
- The prototypes of the *chmod* and *fchmod* File APIs are as follows:

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

int chmod(const char* path_name, mode_t flag);
int fchmod(int fdesc, mode_t flag);
```

- *path_name* – path name of an existing file.
- *fdesc* – file descriptor of a file.
- *flag* – contains new access permission and special flags to be set on the file.
- *flag* value is the same as that of *open* API.

- It can be specified as an octal integer value in UNIX, or constructed from the manifested constants defined in the <sys/stat.h> header.
- The access permission specified in the *flag* argument of *chmod* is not modified by the calling process umask.
- The following function turns on the *set-UID* flag, removes group write permission and others read and execute permission on a file named "*/usr/joe/funny.book*".

```
#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/joe/funny.book", &statv))
                perror("stat");
        else{
                flag = (statv.st_mode & ~flag)|S_ISUID;
                if(chmod("/usr/joe/funny.book", flag))
                        perror("chmod");
        }
}
```

- It first calls *stat* to get the current access permission of the respective file, then it masks off the group write permission and others read and execute permissions from the *statv.st_mode*.
- Then it sets the *set-UID* flag in the *statv.st_mode*.
- All the other existing flags are unmodified.
- The final *flag* value is passed to *chmod* to carry out the changes on the file.
- If either the *chmod* or *stat* call fails, the program will call *perror* to print a diagnostic message.

(XV) chown, fchown, lchown File APIs

- *chown, fchown* functions are used to change user ID and group ID of files.
- The file whose IDs have to be changed is referred by path name or file descriptor.
- *chown* and *chgrp* UNIX commands are implemented based on these APIs.
- *lchown* function changes the ownership of the symbolic link file.

- The prototypes of the *chown* and *fchown* File APIs are as follows:

```
#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* – path name of an existing file.
- *fdesc* – file descriptor of a file.
- *uid* – new user ID to be assigned to the file.
- *gid* – new group ID to be assigned to the file.
- If actual value of the *uid* or *gid* argument is -1, corresponding ID of the file is not changed.
- In BSD UNIX, only a process with superuser privilege can use these functions to change any file user ID or group ID.
- If a process effective user ID matches a file user ID and its effective group ID matches the file group ID, the process can change the file group ID only.
- However, in UNIX System V, if a process effective user ID matches either a file user ID or superuser user ID, the process can change both file user ID and file group ID.
- If *chown* is called by a process that has no superuser privileges and it succeeds, it will clear the file set-UID and set-GID flags.
- If *chown* is called by a process with the effective UID of a superuser, it is implementation-dependent as to how *chown* will treat the set-UID and set-GID flags of the files it modifies.
- The following program implements the *chown* UNIX command.
- It takes at least two command line arguments – the first one is a user name to be assigned to files, and the second (or any subsequent) arguments are file path names.
- The program first converts a given user name to a user ID via the *getpwuid* function.
- If that succeeds, the program processes each named as follows – it calls *stat* to get the file group ID, then calls *chown* to change the file user ID.
- If either the *stat* or *chown* API fails, *perror* will be called to print a diagnostic message.

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

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

(XVI) utime File API

- *utime* function modifies the access and modification timestamps of a file.
- The prototype of the *utime* File API is:

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

int utime(const char* path_name, struct utimbuf* times);
```

- *path_name* – path name of an existing file.
- *times* – specifies new access time and modification time for the file.
- *struct utimbuf* is defined in the <utime.h> header in POSIX.1, but in <sys/types.h> in UNIX System V.

```
struct utimebuf{
        time_t actime;
        time_t modtime;
};
```

- If *times* is specified as 0, the API will set the named file access time and modification time to the current time.
- If *times* is an address of a variable of the type , the API will set the named file access time and modification time according to the values specified in the variable.

- Return value of *utime* is 0 if it succeeds or -1 if it fails.
- Possible causes of failures:
  - *path_name* argument is invalid.
  - Process has no access permission and ownership to a named file.
  - *times* argument has invalid address.
- The following program uses the *utime* function to change the access and modification timestamps of files. The timestamp to set is also defined by users.

```
#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 || sscanf(argv[1], "%d", &offset) != 1){
                cerr << "Usage:" << argv[0] << "<offset> <file>\n";
                return 1;
        }
        times.actime = time.modtime = time(0) + offset;
        for(-i=1; i<argc; i++)
                if(utime(argv[i], &times))
                        perror("utime");
        return 0;
}
```

- The program defines a variable called *times* of *struct utimbuf* and initializes it with the current time value (as obtained from the *time* function call) and a user-defined offset time (in seconds).
- The subsequent command line arguments to the program should be one or more file path names.
- For each of these files, the program will call *utime* to update the file access time and modification time.
- If any *utime* call fails, the program will call *perror* to print a diagnostic message.

(XVII) File and Record Locking

- UNIX systems allow multiple processes to read and write the same file concurrently.
- This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when data in a file can be overridden by another process.
- It is important for applications like database manager, where no other process can write or read a file while a process is accessing a database file.

- To remedy 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 write or read lock on either a portion of a file or an entire file.
- Write lock – prevents other processes from setting any overlapping read / write locks on the locked region of a file (*exclusive lock*).
- Read lock – prevents other processes from setting any overlapping write locks on the locked region of a file (*shared lock*).
- File locks are mandatory if they are enforced by an operating system kernel.
- If a mandatory exclusive lock is set on a file, no process can use the *read* / *write* system calls to access data in 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.
- File locking is a mechanism used to synchronize reading and writing of shared files by multiple processes.
- If a process locks up a file, other processes that attempt to write to the locked regions are blocked until the former process releases its lock.
- Mandatory locks may cause problems.
- If a runaway process sets a mandatory exclusive lock on a file and never unlocks it, no other process 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, but BSD UNIX and POSIX systems do not.
- If a file lock is not mandatory, it is an advisory lock.
- Advisory lock is not enforced by a kernel at the system call level.
- Even though a read / write lock is set on a region of a file, other processes can still use the *read* / *write* APIs to access the file.
- Procedure followed for every read / write operation using advisory locks –
  - Set a lock at the region to be accessed. If this fails, process can wait for request to become successful or try later.
  - Read / write locked region after a lock is acquired successfully.
  - Release the lock.
- Advisory lock on a region of a file will not violate any lock protection set by other processes on the same region.
- Other process will also not modify that region when the lock is imposed.
- Process should release the lock as soon as it is done, thereby allowing access to other processes.

- Advisory lock is considered safe, as no runaway processes can lock up any file forcefully.
- Other processes can go ahead and read / write a file after a fixed number of failed attempts to lock the file.
- *Drawback* – programs that create processes to share files must follow the file locking procedure to be co-operative.
- UNIX System V and POSIX.1 use the *fcntl* API for file locking.

```
#include<fcntl.h>

int fcntl(int fdesc, int cmd, ...);
```

- API can be used to impose read / write locks on either a segment or an entire file.
- *cmd_flag* values in *fcntl* API are used for file locking.

cmd_flag	**Significance**
F_SETLK	Sets a file lock. Does 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

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

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

***l_type* value**	**Significance**
F_RDLCK	Sets 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

***l_whence* value**	**Significance**
SEEK_CUR	*l_start* value is added to current file pointer address
SEEK_SET	*l_start* value is added to byte 0 of the file
SEEK_END	*l_start* value is added to end (current size) of the file

- The lock set by *fcntl* API is advisory lock.
- POSIX does not support mandatory locks.

- UNIX System V.3 and V.4 support mandatory locks using *fcntl* API:
  - Turn ON the set-GID flag of the file
  - Turn OFF the group execute access right of the file
- *chmod* can also be used to set mandatory read or write locks on a file, using:

```
chmod a+l <file_name>
```

- All file locks set by a process will be unlocked when the process terminates.
- If a process locks a file and creates a child process via *fork*, then the child process will not inherit the file lock.
- Return value of *fcntl* is 0 if it succeeds or -1 if it fails.
- Possible causes of failures of *fcntl* API:
  - File descriptor is invalid.
  - Requested region conflicts with locks set by another process.
  - Invalid data in third argument.
  - Maximum number of locks per file has been reached.
- The following program illustrates how *fcntl* is used for file locking.

```
#include<iostream.h>
#include<stdio.h>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
int main(int argc, char* argv[]){
        struct flock fvar;
        int fdesc;
        while(--argc > 0){
                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;
                while(fcntl(fdesc, F_SETLK, &fvar) == -1){
                        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;
                        }
                }
                fvar.l_type = F_UNLCK;
                fvar.l_whence = SEEK_SET;
                fvar.l_start = 0;
                fvar.l_len = 0;
                if(fcntl(fdesc, F_SETLKW, &fvar) == -1)
                        perror("fcntl");
        }
        return 0;
}
```

- The above program takes one or more path names as arguments.
- For each file specified, the program attempts to set an advisory lock on the entire file via *fcntl* API.
- If the *fcntl* call fails, the program scans the file to list all lock information to the standard output.
- Specifically, for each locked region, the program reports the following:
  - the file path name
  - the PID that locks that region
  - the start address of the locked region
  - the length of the locked region
  - whether the lock is exclusive (*w*) or shared (*r*)
- The program loops repeatedly until a *fcntl* call succeeds in locking the file.
- After that, the program will process the file in some way, then it calls *fcntl* again to unlock the file.

(XVIII) Directory File APIs

- Directory file APIs are used to help users in organizing their files into some structure based on the specific use of files.
- These APIs are used by the operating system to convert file path names to their inode numbers.
- Directory files are created in BSD UNIX and POSIX using *mkdir* API.

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

int mkdir(const char* path_name, mode_t mode);
```

- *path_name* – path name of a directory file to be created.
- *mode* – access permission for the owner, group and others to be assigned to the file.
- *mode* value is modified by the calling process umask.
- Return value of *mkdir* is 0 if it succeeds or -1 if it fails.
- Possible causes of failure of *mkdir* API:
  - *path_name* is invalid.
  - Calling process lacks permission to create the specified directory.
  - *mode* is invalid.
- UNIX System V.3 uses *mknod* API to create directory files.
- UNIX System V.4 supports both *mkdir* and *mknod* APIs to create directory files.
- Difference – *mknod* does not contain . (current directory) and .. (parent directory)

- On systems that do not support *mkdir* API, the directories can be created using *system* API as follows:

```
char syscmd[256];
sprintf(syscmd, "mkdir %s", <directory_name>);
if(system(syscmd) == -1)
        perror("mkdir");
```

- For a newly created directory, the user ID is set to effective user ID of the calling process.
- The group ID is set to either the effective group ID of the calling process or group ID of the parent directory that hosts new directory.
- A directory is a record-oriented file, where each record stores a file name and inode number of file that resides in that directory.
- To allow a process to scan directories in a file system-independent manner, a directory record is defined as *struct dirent* in the <dirent.h> header for UNIX System V and POSIX.1, and as *struct direct* in the <sys/dir.h> header in BSD UNIX 4.2 and 4.3.
- The *struct dirent* and *struct direct* data types have one common field, *d_name*, which is a character array that contains the name of a file residing in a 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);
```

Function	Use
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.
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 using *rmdir* API.
- Users may also use *unlink* API to remove directories as the superuser.
- These APIs require that directories to be removed should be empty.
- Prototype of *rmdir* function –

```
#include<unistd.h>

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

- The following program illustrates the use of *mkdir, opendir, readdir, closedir* and *rmdir* APIs.

```
#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){
              if(!(dir_fdesc = opendir(*++argv))){
                     if(mkdir(*argv, S_IRWXU|S_IRWXG|S_IRWXO) == -1)
                            perror("opendir");
                     continue;
              }
              for(int i=0; i<2; i++){
                     for(int count=0; dp=readdir(dir_fdesc);){
                            if(i)
                                   cout << dp->d_name << endl;
                            if(strcmp(dp->d_name, ".") && strcmp(dp->d_name, ".."))
                                   count++;
                     }
                     if(!count){
                            rmdir(*argv); break;
                     }
                     rewinddir(dir_fdesc);
              }
              closedir(dir_fdesc);
       }
}
```

The above program takes one or more directory file path names as arguments.

- For each argument, the program opens it via the *opendir* and stores the file handler in the *dir_fdesc* variable.
- If the *opendir* call fails, the program assumes the directory does not exist and attempts to create it via the *mkdir* API.
- If the *opendir* succeeds, the program scans the directory file using the *readdir* API and determines the number of files, excluding the "." and ".." files in the directory.
- After this is done, it removes the directory via the *rmdir* API, if it is empty.
- If directory is not empty, then the program resets the file pointer associated with the *dir_fdesc* via the *rewinddir*, it then scans the directory file a second time and echoes all file names in that directory to the standard output.
- When the second round of directory scanning is completed, the program closes the *dir_fdesc* with the *closedir* API.

(XIX) Device File APIs

- Device file APIs are used to interface physical devices (console, modem, floppy) with application programs.
- When a process reads/writes to a device file, kernel uses major and minor device numbers of a file to select a device driver function to carry out the actual data transfer.
- These device files may be character-based or block-based.
- Device file support is implementation-dependent.
  POSIX does not specify how device files are to be created.
- 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);
```

- *path_name* – path name of a device file to be created.
- *mode* – access permission for the owner, group and others to be assigned to the file.
- *mode* value is modified by the calling process umask.
- *device_id* – contains major and minor device numbers.
- Return value of *mknod* is 0 if it succeeds or -1 if it fails.
- Possible causes of failure of *mknod* API:
  - *path_name* is invalid.
  - Calling process lacks permission to create a device file.
  - *mode* is invalid.
- *mknod* File API must be called by a process with superuser privileges.

- User ID and Group ID attributes of a device file are assigned in the same manner as for regular files.
- File size attribute of any device file has no meaningful use.
- Once a device file is created, any process may use the *open* API to connect to the file.
- It can then use *read*, *write*, *stat* and *close* APIs to manipulate the file.
- *lseek* is applicable to block device files, but not to character device files.
- Device file may be removed via the *unlink* API.
- When a process calls *open* API to establish connection with a device file, the O_NONBLOCK and O_NOCTTY flags maybe specified.
- If calling process has no terminal and it opens a character device file, the kernel will set the device file as controlling terminal (O_NOCTTY flag).
- Non-block flag (O_NONBLOCK flag) specifies that *open* call and any subsequent read/write calls to a device file should be non-blocking to a process.
- Only privileged users (superuser) may use the *mknod* API to create device files.
- All other users may read and write device files as if they were regular files, subjected to access permissions set on those device files.
- Treatment of device files is almost identical to that of regular files, except the way that device files are created and that *lseek* is not applicable for character device files.
- An example to create a block device file called *SCSI5* with major and minor device numbers of 15 and 3, and access rights of read-write-execute for everyone, using *mknod* system call is given as follows –

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

- The following program illustrates the use of *mknod, open, read, write* and *close* APIs on a block device file.
- The program takes three arguments: a device file name, a major device number and a minor device number.
- The program will use these arguments to create a character device file using *mknod*.
- The program opens the file for read-write and sets the O_NONBLOCK and O_NOCTTY flags.
- It then reads data from the device file and echoes data to the standard output.
- When end-of-file is encountered, the program closes the file descriptor associated with the device file and terminates.

```
#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);
}
```

(xx) FIFO File APIs

- FIFO Files are also known as *named pipes*.
- They are special pipe device files used for interprocess communication.
- Any process can attach to a FIFO file to read, write, or read-write data.
- Data written is stored in fixed-size buffer and retrieved in first-in-first-out (FIFO) order.
- FIFO files are created in BSD UNIX and POSIX using *mkfifo* API:

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

int mkfifo(const char* path_name, mode_t mode);
```

- *path_name* – path name of a FIFO file to be created.
- *mode* – access permission for the owner, group and others to be assigned to the file, as well as S_IFIFO flag to indicate FIFO file.
- *mode* value is modified by the calling process umask.
- Return value of *mkfifo* is 0 if it succeeds or -1 if it fails.
- Possible causes of failure of *mkfifo*:
  - *path_name* specified is invalid.

- Calling process lacks permission to create the FIFO file.
- *mode* argument is invalid.

- An example to create a FIFO file called *FIFO5* with access permission of read-write-execute for everyone using *mkfifo* call is given as follows:

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

- UNIX System V.3 uses *mknod* API to create FIFO files.
- UNIX System V.4 supports *mkfifo* API to create FIFO files.
- User ID and Group ID attributes of a FIFO file are assigned in same manner as that for regular files.
- Once a FIFO file is created, any process may use the *open* API to connect to the file.
- It can then use *read*, *write*, *stat* and *close* APIs to manipulate the file.
- *lseek* is not applicable to FIFO files.
- FIFO file may be removed via the *unlink* API.
- If a process opens FIFO file for read-only, the kernel blocks the process until there is another process that opens the same file for write operation.
- If a process opens FIFO file for write, the kernel blocks the process until there is another process that opens the same file for read operation.
- This provides process synchronization.
- If a process writes to FIFO file that is full, the kernel blocks the process until there is another process has read data from FIFO file to make room for new data in the FIFO file.
- If a process reads from FIFO file that is empty, the kernel blocks the process until there is another process that writes data to the FIFO.
- If a process desires to not be blocked by FIFO file, the O_NONBLOCK flag should be specified in *open*.
- If process subsequently calls *read* or *write* API on FIFO file and data is not ready for transfer, functions will return -1 values.
- UNIX System V defines O_NDELAY flag which is similar to O_NONBLOCK flag. Difference – functions will return 0 value when blocking a process
- If a process writes to FIFO file that has no other process attached to it for read, the kernel will send SIGPIPE signal to the process to notify it of illegal operation.
- If a process reads from FIFO file that has no other process attached to it for write, the process will read remaining data in the FIFO and then specify end-of-file indicator.
- If two processes have to communicate via a FIFO file, then it is necessary for the writer process to close its file descriptor when it is done, so that the reader process can see the end-of-file condition.
- It is possible for process to open FIFO file for both read and write.
- POSIX.1 does not specify how to handle this, but UNIX systems will not block the process.

- The process can use the file descriptor returned from *open* API to read and write data with the FIFO file.
- The following program illustrates the use of *mkfifo*, *open*, *read*, *write* and *close* APIs for a FIFO file.

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

int main(int argc, char* argv[]){
        if(argc != 2 && argc != 3){
                cout << "Usage:" << argv[0] << "<file> [<arg>]\n";
                return 0;
        }
        int fd;
        char buf[256];
        (void)mkfifo(argv[1], S_IFIFO|S_IRWXU|S_IRWXG|S_IRWXO);
        if(argc == 2){
                fd = open(argv[1], O_RDONLY|O_NONBLOCK);
                while(read(fd, buf, sizeof(buf)) == -1 && errno == EAGAIN)
                        sleep(1);
                while(read(fd, buf, sizeof(buf)) > 0)
                        cout << buf << endl;
        }
        else{
                fd = open(argv[1], O_WRONLY);
                write(fd, argv[2], strlen(argv[2]));
        }
        close(fd);
}
```

- The above program takes one or two arguments.
- The first argument is the name of a FIFO file to be used.
- The program uses *mkfifo* to create the FIFO file if it does not exist.
- It then checks whether it has one or two arguments.
- If it has one argument, it will open the FIFO file for read-only.
- It will then read all data from the FIFO file and echo them to the standard output.
- However, if the process has two arguments, it will open the FIFO file for write and will write the second argument to the FIFO file.
- This program can therefore be run twice to create two processes that communicate through a FIFO file.
- Another way to create FIFO files is by using the *pipe* API.

- Prototype of *pipe* function is given below –

```
#include<unistd.h>

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

- Transient FIFO file created by *pipe* refers to the fact that there is no file that is actually created in file system; it is just discarded once all processes close their file descriptor that references the FIFO file.
- Uses of fds argument –
  fds[0] – file descriptor to read data from FIFO file.
  fds[1] – file descriptor to write data to FIFO file.
- FIFO file can't be referenced by path name and its use is restricted to related processes.
- *Example*: parent process creates FIFO file, which then creates child processes, who inherit the FIFO file descriptors.
  They communicate amongst themselves and with the parent via the FIFO file.

(XXI) Symbolic Link File APIs

- Defined in BSD UNIX 4.2 and used in BSD UNIX 4.3, System V.3 and V.4.
- These were developed to overcome shortcomings of hard links:
  - Can link files across file systems.
  - Can link directory files.
  - Always reference the latest version of the files to which they link.
- Hard links can be broken by removal of one or more links.
- Symbolic links are not broken by removal of one or more links; instead, the link is re-established with the new file.
- Example – even if the file */usr/go/test1* is removed when there is a symbolic link, the new file of the same name can be linked again, but it is not possible with a hard link.

```
ln /usr/go/test1 /usr/joe/hdlink

ln -s /usr/go/test1 /usr/joe/symlnk
```

- Prototype of symbolic link File API is given as follows –

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

int symlink(const char* org_link, const char* sym_link);
int readlink(const char* sym_link, char* buf, int size);
int lstat(const char* sym_link, struct stat* statv);
```

- *org_link* – original file path name.
- *sym_link* – symbolic link path name to be created.
- For example, the *symlink* call to create a symbolic link called */usr/joe/lnk* for a file called */usr/go/test1* is given as:

```
symlink("/usr/go/test1", "/usr/joe/lnk");
```

- Syntax of *symlink* is the same as that of *link* API.
- It is being proposed to be included in POSIX.1 standard.
- Return value of *symlink* is 0 if it succeeds or -1 if it fails.
- Possible causes of failure of *symlink*:
  - path name specified is illegal.
  - Calling process lacks permission to create the new file.
  - *sym_link* file already exists.
- *readlink* is used to query the path name to which a symbolic link refers.
- *sym_link* – symbolic link path name.
- *buf* – character array buffer that holds the return path name referenced by the link.
- *size* – maximum capacity (bytes) of the *buf* argument.
- Return value of *readlink* is actual number of characters of a path name placed in the *buf* argument or -1 if it fails.
- Possible causes of failure of *readlink*:
  - Calling process lacks permission to access the symbolic link file.
  - *sym_link* path name is not symbolic link.
  - *buf* argument is an illegal address.
- 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* is used to query the file attributes of symbolic links.
- Function prototype and return values of *lstat* are same as that of *stat*.
- *lstat* can also be used on non-symbolic link files, and it behaves like *stat*.
- *ls –l* command uses *lstat* API to display information of all file types.
- The following program emulates the UNIX *ln* command.

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

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]);
        else
                return link(argv[1], argv[2]);
}
```

- The main function of the program is to create a link to a file.
- The names of the original file and the 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.

(XXII) UNIX Processes

- A *program* is an executable file residing on disk in a directory.
- A *process* is an executing instance of a program.
- The process also known as *task* in some OS.
- Every process has a unique numeric identifier called the *process ID (PID)*.
- PID is always a non-negative integer.
- Three primary functions are present for process control in the UNIX systems – *fork*, *exec* and *waitpid*.
- Process usually has one thread of control, i.e., one set of machine instructions executing at a given time.
- Multiple threads of control are also available, wherein parallelism is possible on multiprocessor systems.
- All threads within a process share the same address space, file descriptors, stacks and process-related attributes.
- As they can access the same memory, threads need to synchronize access to shared data among themselves to avoid inconsistencies.
- Threads are identified by unique thread IDs.

- Thread IDs are local to a process, i.e., thread ID from one process has no meaning in another process.
- Functions to control threads are similar to those used to control processes.

(XXIII) Process Environment

- Here, the environment of a single process is examined.
- Descriptions of main function, command line arguments are given.
- Typical memory layout, additional memory allocations are explored.
- Use of environment variables and ways to terminate processes are also seen.

(XXIV) main Function

- The execution of a C program starts with *main*.
- Prototype for *main* function is given as follows –

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

- *argc* - number of command line arguments.
- *argv* - array of pointers to the command line arguments.
- When C program is executed by the kernel, a special startup routine is called before *main*.
- The executable program file specifies this routine as starting address for the program.
- This is set up by link editor when it is invoked by the C compiler.
- Routine takes two values from kernel for setup – command line arguments and environment.

(XXV) Process Termination

- There are eight ways for a process to terminate in the UNIX system.
- There are five normal ways for process termination:
  - Return from *main*
  - Calling *exit*
  - Calling *_exit* or *_Exit*
  - Return of the last thread from its start routine
  - Calling *pthread_exit* from the last thread
- There are also three abnormal ways for process termination:
  - Calling *abort*
  - Receipt of a signal
  - Response of the last thread to a cancellation request

- The start-up routine is written such that if *main* function returns, *exit* is called.
- Three functions terminate a function normally –
  - *_exit* and *_Exit* return to kernel instantly
  - *exit* performs cleanup processing and then returns to the kernel.

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

- *exit* function always performs a clean shutdown of standard I/O library.
- *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 – *exit status*.
- Exit status of process is undefined when –
  - Any of the exit functions are called without an exit status.
  - *main* does a return without a return value.
  - *main* is not declared to return an integer.
- If return type of *main* is integer and *main* has an implicit return, exit status of process is 0.
- Returning an integer value from *main* is equivalent to calling *exit* with the same value.

  exit(0); is same as **return(0);** from the *main* function.
- Classic "hello, world" example gives different exit status codes on different systems.

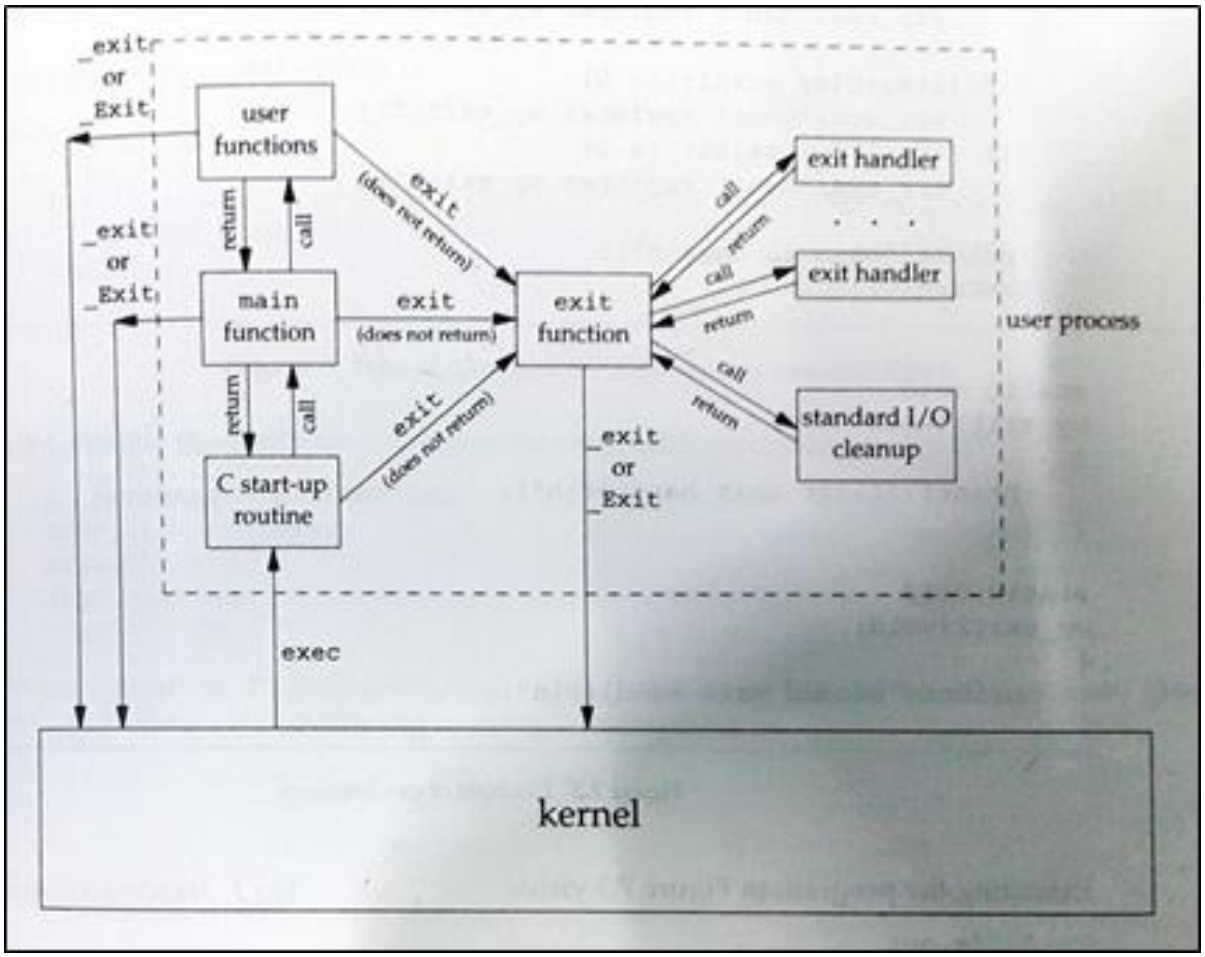
```
#include<stdio.h>

main(){
        printf("hello, world\n");
}
```

- The exit status depends on contents of stack and register contents at the time of return from *main*.
- *atexit* function is a function that is automatically called by *exit*.
- These types of functions are known as *exit handlers*.
- ISO C processes can register up to 32 such functions to operate as exit handlers.

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

- It returns 0 if successful, non-zero on error.
- Address of a function is passed as the argument to *atexit*.
- *exit* calls these functions in reverse order of their registration.
- Each function is called as many times as it was registered.

- ISO C and POSIX.1 – *exit* first calls the exit handlers and then closes all open streams via *fclose* function.
- POSIX.1 extends ISO C standard by specifying that any exit handlers installed will be cleared if the program calls any of the *exec* family of functions.
- The following figure summarizes how a C program is started and the various ways in which it can terminate.



- 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.
- A process can also be involuntarily terminated by a signal.
- The following program demonstrates the use of *atexit* function.
- An exit handler is called once for each time it is registered.

- *exit* is not used, instead the program directly returns from *main*.

```
#include "apue.h"

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

int main(void){
        if(atexit(my_exit2) != 0)
                err_sys("Can't register my_exit2");
        if(atexit(my_exit1) != 0)
                err_sys("Can't register my_exit1");
        printf("Main function completed.\n");
        return(0);
}

static void my_exit1(void){
        printf("First exit handler.\n");
}

static void my_exit2(void){
        printf("Second exit handler.\n");
}
```

(XXVI) Command Line Arguments

- When a program is executed, the process that invokes *exec* can pass command line arguments to the program.
- This is part of the normal operation of the UNIX system shells.
- *Example code* – output all command line arguments to standard output.

```
#include "apue.h"

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

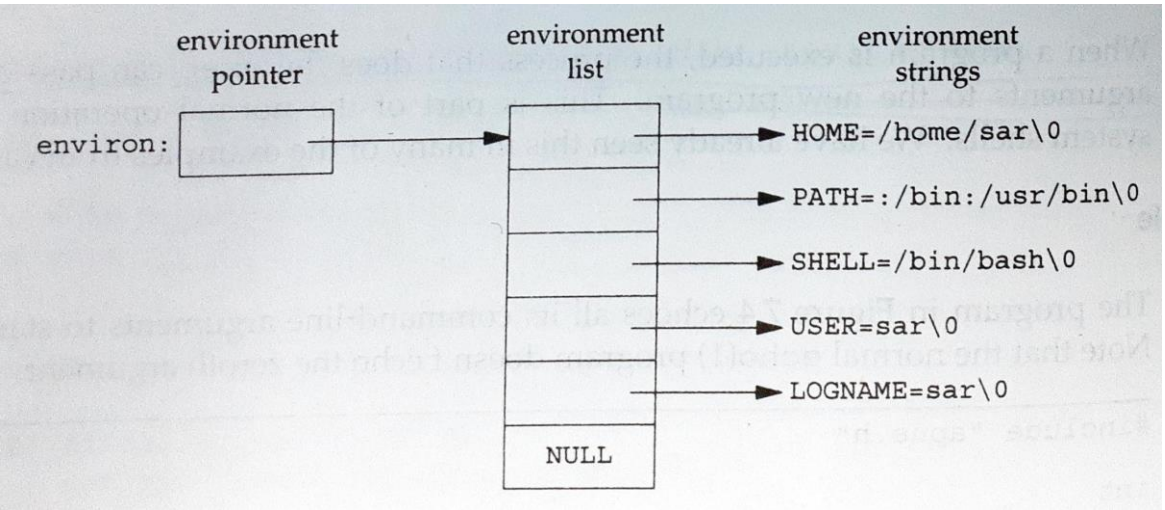
- The normal *echo* command does not output the zeroth command line argument.

(XXVII) Environment List

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

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

- Example – if 5 strings are present in the environment, it would look like the figure below.



- Null bytes are shown at the end of each string in the environment.
- Here, the environment pointer – *environ*,
  environment list – array of pointers, and
  environment strings – strings pointed by the environment list.
- By convention, the type of strings in the environment are:

```
name=value
```

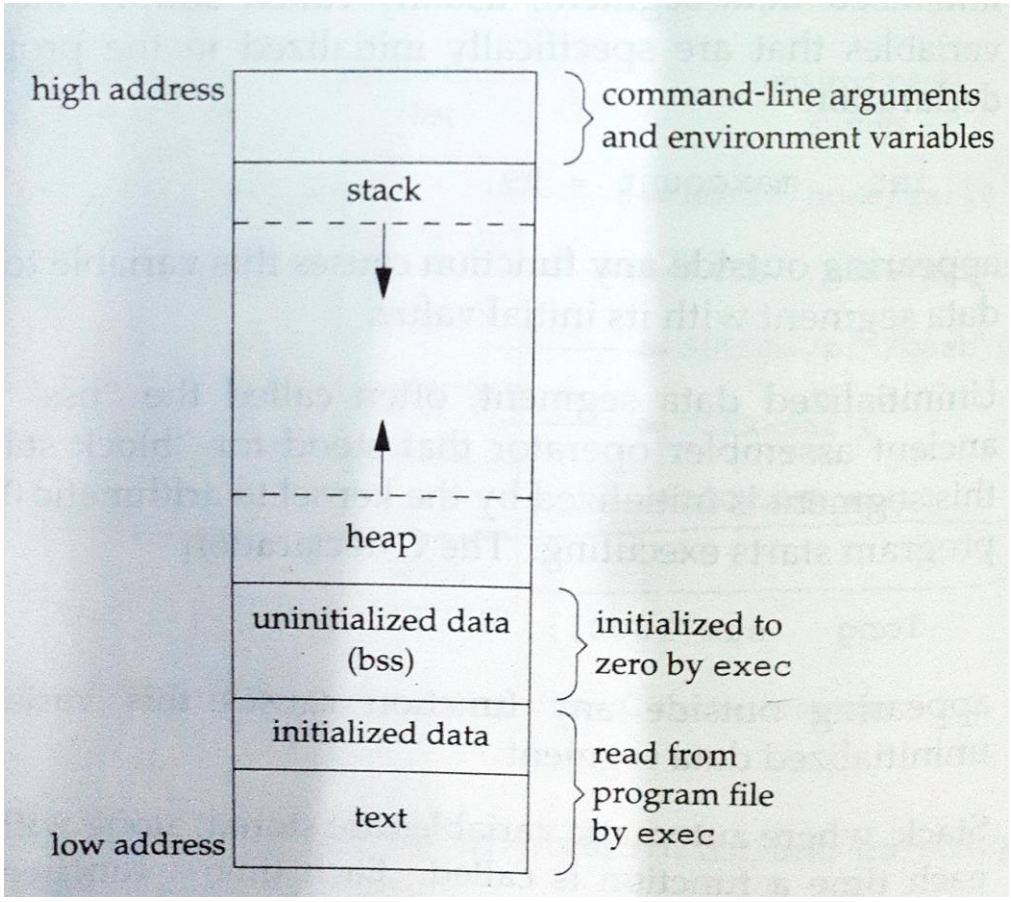
- Most pre-defined names are uppercase.
- Historic UNIX systems also contained a third argument to *main* function that was the address of the environment list.

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

- ISO C specifies *main* has two arguments, stating that the third argument has no benefit over the existing global variable *environ*.
- POSIX.1 specifies environ should be used instead of the third argument.
- Specific environment variables accessed using *getenv* and *putenv* functions, but *environ* necessary to go through the entire environment.

(XXVIII) Memory Layout of a C Program

- The following figure gives the typical memory arrangement of a C program.



- The main segments in a C program are –
  - Text segment
  - Initialized data segment
  - Uninitialized data segment
  - Stack
  - Heap
- Additional segments are – symbol table, debugging information, linkage tables for dynamic shared libraries.
  These are not loaded as part of program image executed by process.
- *Text segment* consists of machine instructions executed by CPU.
- It is usually sharable so that a single copy needs to be in memory for frequently executed programs such as text editors, C compilers and shells.
- It is often read-only, so as to prevent a program from accidentally modifying its instructions.
- *Initialized data segment (a.k.a. the data segment)* consists of variables that are specifically initialized in the program.

- *Uninitialized data segment (a.k.a the bss segment)* consists of data that is initialized by the kernel to arithmetic 0 or null pointers before program starts executing.
- *Stack* is used for storage of automatic variables and information of each function call.
- Address of where the function has to return to, information about function caller's environment (machine registers) are stored on the stack.
- Newly called function allocates room on the stack for automatic and temporary variables.
- Recursive functions in C utilize the stack. A new stack frame is used each time it calls itself, so that one set of variables does not interfere with variables from another instance.
- *Heap* is used for the allocation of dynamic memory.
- It is located between uninitialized data and stack.
- Stack growth is from higher numbered addresses to lower numbered addresses.
- Unused virtual space between the top of heap and the top of stack is large.
- Contents of uninitialized data segment are not stored in the program file on disk. This is because the kernel sets it to 0 before program starts running.
- The portions of program to be saved in program file are text segment and initialized data.
- ***size*** command can be used to obtain sizes (in bytes) of text, data and bss segments.

(XXIX) Shared Libraries

- Most UNIX systems support shared libraries.
- Arnold described early implementation under System V, and Gingell described different implementation under SunOS.
- Shared libraries remove common library routines from executable file and maintain a single copy of the library routine in memory that is referred by all processes.
- It reduces the size of each executable file but adds runtime overhead.
  - when program is first executed.
  - first time each shared library function is called.
- Advantage – library functions can be replaced with new versions without having to re-link and edit every program that uses library.
  (The assumption is that the number and type of arguments are not changed.)
- Different systems have different ways for a program to use or not use shared libraries.
- There is a massive decrease in file size when program uses shared libraries.

(XXX) Memory Allocation

- ISO C specifies 3 functions for memory allocation.
- *malloc* allocates a specified number of bytes of memory.
  Initial value of memory is indeterminate.
- *calloc* allocates space for a specified number of objects of a specified size.
  Space initialized to all 0 bits.
- *realloc* increases or decreases the size of a previously allocated area.
  If the size increases, the previously allocated area is moved somewhere else to provide additional room at the end.
  Initial value of space between old contents and new area is indeterminate.
- Prototypes of the memory allocation functions:

```
#include<stdlib.h>

void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
void free(void *ptr);
```

- Return values of these functions – non-null pointer if successful, NULL on error.
- Pointer returned by the 3 allocation functions can be used for any data object.
- Generic *void* * pointers are returned, therefore explicit type casting of the pointer to other data types is not necessary.
- *free* function causes the space pointed to by *ptr* to be deallocated.
- This free space is put into a pool of available memory and can be allocated later.
- If there is no room for *realloc* beyond existing region, then it allocates new area that is large enough, copies old data, frees old area and returns pointer to new area.
- If *ptr* is null pointer, then *realloc* behaves like *malloc* and allocates a region of specified size.
- Allocation routines are implemented by *sbrk* system call.
- It expands or contracts the heap (memory) of the process.
- It is not possible to decrease memory size using *malloc* and *free* – only reallocation of freed space to a later process (not returned to kernel, kept in *malloc* pool) is possible.
- Most implementations allocate more space than is requested and the additional space is used for record keeping.
- Record keeping involves storing details such as the size of allocated block, pointer to next allocated block.
- Writing past the end (or before the start) of allocated area can overwrite record keeping information and cause catastrophic data corruption errors.

- If memory before and/or after the record keeping are used for other processes, then it becomes even more difficult to find the source of data corruption.
- Other possible fatal errors include freeing a block that was already freed, calling *free* with a pointer that was not obtained from one of the 3 *alloc* functions.
- *Leakage* refers to a situation when a process calls *malloc*, but forgets to call *free* afterwards. This causes the memory usage to continually increase.
- The size of process's address space slowly increases until no free space is left.
- Therefore, performance degradation occurs due to excessive paging overhead.

- There are a few alternate memory allocators – *libmalloc*, *vmalloc*, *quick-fit*, *alloca* functions.
- *libmalloc* – provides a set of interfaces matching the ISO C memory allocation functions.
  mallopt function allows a process to set certain variables that control the operation of the storage allocator.
  mallinfo function provides statistics on the memory allocator.
- *vmalloc* - allocation of memory using different techniques for different regions of memory.
  Provides emulations of the ISO C memory allocation functions.
- *quick-fit* – standard *malloc* uses either best-fit or first-fit memory allocation strategy. This is faster than both, but uses more memory.
  Memory is split up into buffers of various sizes and unused buffers are maintained on different free lists, depending on the size of the buffers.
- *alloca* – has the same calling sequence as *malloc*, but memory is allocated from the stack frame of the current function instead of allocating memory from the heap.
  Increases the size of the stack frame.
  Advantage – no need to free the space, goes away automatically when the function returns.
  Disadvantage - some systems cannot support *alloca* if it is not possible to increase the size of the stack frame after the function has been called.

(XXXI) Environment Variables

- Some environment variables are set automatically at login, others have to be set manually.
- Environment variables are set in a shell start-up file to control the shell's actions.
- Function used to fetch values from the environment is *getenv* and given as follows –

```
#include<stdlib.h>

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

- It returns pointer to *value* associated with *name* if successful, NULL if not found.
- Some environment variables are defined by POSIX.1 in the Single UNIX Specification, whereas others are defined only if the XSI extensions are supported.
- ISO C does not define any environment variables.
- The following table gives a few environment variables.

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

- There are also a few environment list functions that can be used to manipulate environment variables - *putenv, setenv, unsetenv, clearenv*.

```
#include<stdlib.h>

int putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

- These functions return 0 if successful, non-zero on error.
- *putenv* takes string of the form *name=value* and places it in the environment list. If *name* already exists, old definition is first removed.

- *setenv* - sets *name* to *value*.
  If *name* already exists in the environment,
  - if *rewrite* is non-zero, existing definition for *name* is first removed.
  - if *rewrite* is 0, existing definition for *name* is not removed, *name* is not set to new *value*, and no error occurs.
- *unsetenv* - removes any definition of *name*.
  It is not an error is such a definition does not exist.
- *clearenv* - remove all entries from the environment list.

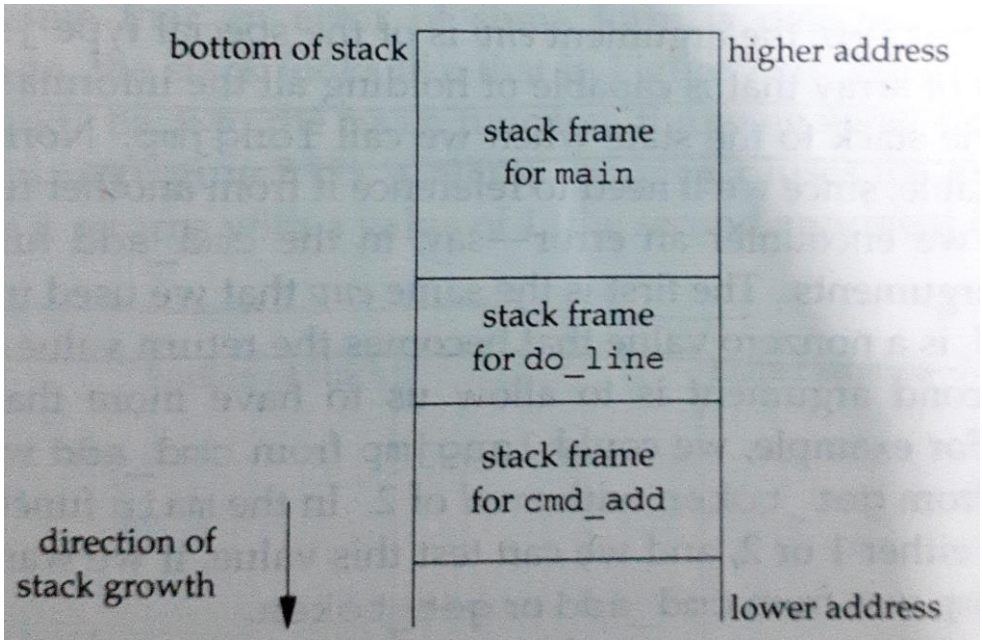
- Deleting an environment string is simple – find pointer in the environment list and move all subsequent pointers down one.
- Adding new string or modifying an existing string is difficult because:
  - top of stack cannot be expanded upward (it is at the top of address space of the process).
  - cannot be expanded downward (all stack frames below it cannot be moved).
- Modifying an existing *name* –
  - If size of new *value* <= size of existing *value*, copy new string over the old string.
  - If size of new *value* > size of existing *value*, use *malloc* to obtain room for the new string, copy new string to this area and replace old pointer in environment list for *name* with pointer to this newly allocated area.
- Adding a new *name* – *malloc* is called first to allocate room for *name=value* string and copy string to new area.
  - If first time addition – call *malloc* to obtain room for new list of pointers, copy old environment list to new area, store a pointer to *name=value* string at the end of this list (and add null pointer), set *environ* to point to the new list.
  - If not first time addition – already allocated room on the heap anyway, so call *realloc* to allocate room for one more pointer, store a pointer to the *name=value* string at the end of this list (and add null pointer).

(XXXII) setjmp and longjmp Functions

- In C, it is not possible to *goto* a label that is in another function.
- This type of branching can be done using *setjmp* and *longjmp* functions (non-local branching within a process).
- It is useful for handling error conditions that occur in deeply nested function call.
- Typical program skeleton for command processing includes read commands, determine the command type and then call functions to process each command.

```c
#include "apue.h"
#define TOK_ADD 5

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

int main(void){
        char line[MAXLINE];
        while(fgets(line, MAXLINE, stdin)!=NULL)
                do_line(line);
        exit(0);
}
void do_line(char *ptr){
        int cmd;
        tok_ptr = ptr;
        while((cmd = get_token())>0){
                switch(cmd){
                        case TOK_ADD: cmd_add();
                                      break;
                }
        }
}
void cmd_add(void){
        int token;
        token = get_token();
        //some processing
}

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.
- It then calls *get_token* to fetch the next token from the input line.
- First token is assumed to be a command, and *switch* selects each command.
- For the single command, *cmd_add* is called.
  The stack frame is shown as below.



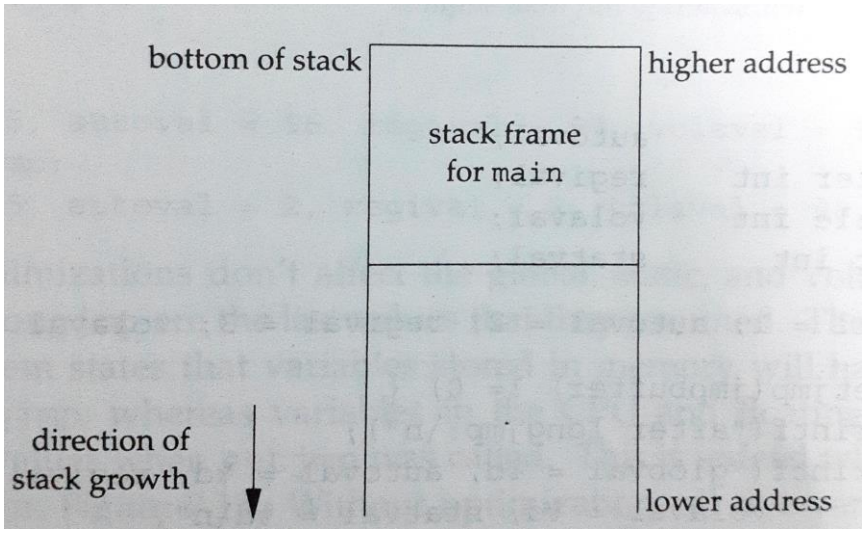
- Storage for the automatic variables is within the stack frame for each function.
  Array *line* is present in the stack frame for *main*.
  Integer *cmd* is present in the stack frame for *do_line*.
  Integer *token* is present in the stack frame for *cmd_add*.
- Non-fatal errors are difficult to handle when the changes are to be made in functions that are deeply nested numerous levels down from *main*.
- Each function with special return values for one level will mess up the program.
- Thus, non-local *goto* is used which is represented by *setjmp* and *longjmp* functions.
- Branching back through the call frames to a function that is in the call path of the current function.

```
#include<setjmp.h>

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

- *setjmp* returns 0 if called directly, non-zero if returning from a call to *longjmp*.
- *setjmp* called from location that we want to return to (here, in the *main* function).
- Here, it returns 0 because it is called directly.
- Argument *env* in *setjmp* is of the special type *jmp_buf*.
  This buffer is an array that can hold all the information required to restore the status of the stack to the state when *longjmp* is called.
- When an error is encountered, *longjmp* is called with two arguments – *env* used in a call to *setjmp* and *val*, a non-zero value that becomes the return value from *setjmp*.
- The changes in the skeleton example when *setjmp* and *longjmp* are used is given below –

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

jmp_buf jmpbuffer;
int main(void){
        char line[MAXLINE];
        if(setjmp(jmpbuffer)!=0)
                printf("error!");
        while(fgets(line, MAXLINE, stdin)!=NULL)
                do_line(line);
        exit(0);
}

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

- When *main* is executed, *setjmp* is called and it records whatever information is needed in the variable *jmpbuffer* and returns 0.
- *do_line* is then called, which calls *cmd_add* and assume that some error is detected.
- Stack frame changes and stack is unwound back to *main*, throwing away the stack frames for *cmd_add* and *do_line*.
- *longjmp* is called, which causes *setjmp* in *main* to return with a value of 1.
  The stack frame then becomes as follows:



(XXXIII) getrlimit and setrlimit Functions

- Every process has a set of resource limits.
- These can be queried and changed by *getrlimit* and *setrlimit* functions.

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

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

- The return value is 0 if successful, non-zero on error.
- These functions are defined as XSI extensions in the Single UNIX Specification.
- Resource limits for a process are established by process 0 when system is initialized and then inherited by each successive process.
- Each call to these functions specifies *resource* and 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
};
```

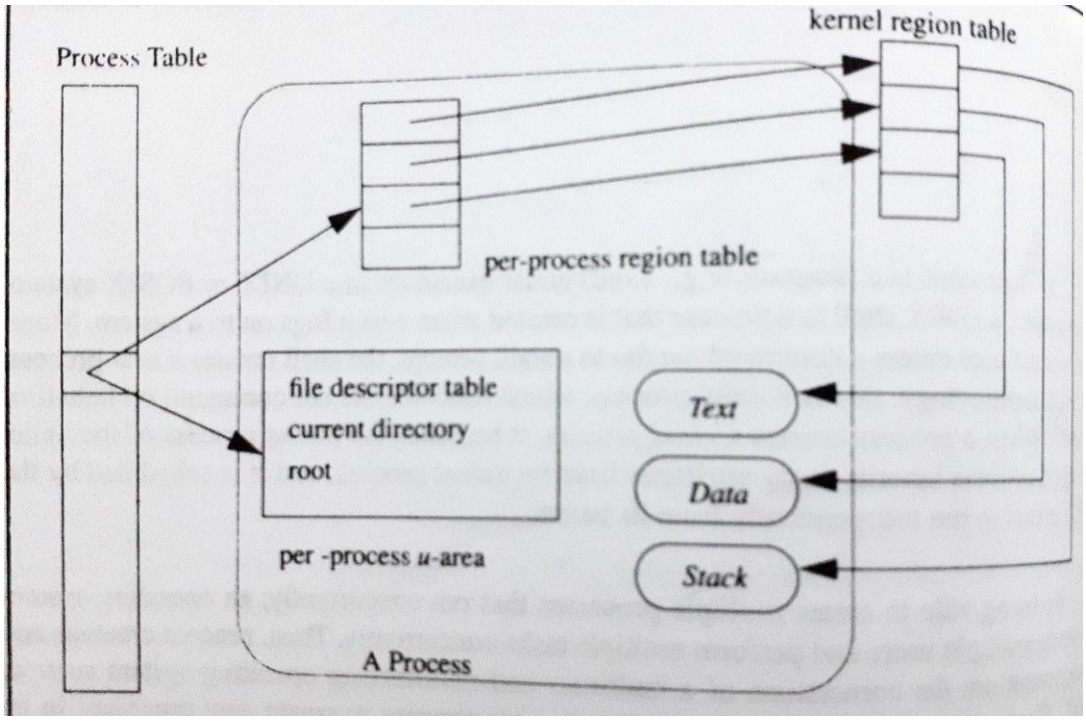
- The rules for changing the resource limits are:
  - Process can change its soft limit to a value <= its hard limit.
  - Process can lower its hard limit to a value >= its soft limit. This is irreversible for normal users.
  - Only a superuser can raise a hard limit.
- Infinite limit is specified by the constant RLIM_INFINITY.
- The following table gives the resource limits that are defined in the UNIX system:

Constant	Description
RLIMIT_AS	Maximum size (in bytes) of total available memory of a process.
RLIMIT_CORE	Maximum size (in bytes) of a core file.
RLIMIT_CPU	Maximum amount of CPU time (in seconds).
RLIMIT_DATA	Maximum size (in bytes) of the data segment.
RLIMIT_FSIZE	Maximum size (in bytes) of a file that may be created.
RLIMIT_LOCKS	Maximum number of file locks a process can hold.
RLIMIT_MEMLOCK	Maximum amount of memory (in bytes) that a process can lock into memory using *memlock()*.
RLIMIT_NOFILE	Maximum number of open files per process.
RLIMIT_NPROC	Maximum number of child processes per real user ID.
RLIMIT_RSS	Maximum resident set size (in bytes).
RLIMIT_SBSIZE	Maximum size (in bytes) of socket buffers that a user can consume at any given time.
RLIMIT_STACK	Maximum size (in bytes) of the stack.
RLIMIT_VMEM	Synonym for RLIMIT_AS.

- Resource limits affect the calling process and are inherited by any of its children.
- Setting of resource limits is built into the shells to affect all future processes.
- Bourne and Korn shells have the *ulimit* command for this purpose.
  C shell has the *limit* command for this purpose.

(XXXIV) Kernel Support for Processes

- Data structure and execution of processes are dependent on OS implementation.



- UNIX Process minimal contents are text, data and stack segments.
- *Segment* is an area of memory that is managed by the system as a unit.
- Text segment is the program text of a process in machine-executable instruction code format.
- Data segment consists of static and global variables and their corresponding data.
- Stack segment refers to the run-time stack. It provides storage for function arguments, automatic variables and return addresses of all active functions for a process at any time.
- UNIX kernel has *process table* which keeps track of all active processes.
- System processes are the processes that belong to the kernel.
- Majority of processes are associated with the users who are logged in.
- Each entry in process table contains pointers to text, data and stack segments and the *u-area* of the process.

- *u-area* is the extension of process table entry and contains other process-specific data (file descriptor table, current root and working directory inode numbers and set of system-imposed process resource limits).
- First process (process 0) is created by the system boot code.
- All other processes are created via the *fork* system call.
- After a *fork* system call, both parent and child processes resume execution at the return of the *fork* function.
- When a process is created by *fork*, it contains duplicated copies of text, data and stack segments of its parent process.
- It also has a file descriptor table that contains references to the same opened files as its parent – both share same file pointer to each opened file.
- After *fork*, parent process may choose to suspend its execution by calling *wait* or *waitpid* system call, until its child process terminates.
- It may continue execution independently of its child process.
  It may use *signal* or *sigaction* function to detect or ignore the child process termination.
- Process terminates its execution by calling *_exit* system call.
  Exit status will be zero if process has executed successfully, or non-zero if it has failed.

(XXXV) Process Control

- Process control includes –
  - creation of new processes
  - program execution
  - process termination
- Various IDs for processes are explored, such as real, effective and saved user IDs and group IDs.
- Interpreter files and *system* function are also explored.
- Process accounting is described in detail as well.

(XXXVI) Process Identifiers

- Every process has unique process ID (PID), which is a non-negative integer.
- It is often used as a piece of other identifiers to guarantee uniqueness.
- *Example* - applications include PID as part of the filename in an attempt to generate unique filenames.
- But PIDs can be reused once a process terminates.

- Reusing PID is usually delayed because new processes should not use PIDs of processes that terminated recently, as there can be confusion between the processes.
- Header file that is mainly used for the process identifiers is ***unistd.h***
- There are 3 special processes in the UNIX system – PID 0, PID 1, PID 2.
- PID 0 is the *scheduler* process (also known as *the swapper*).
  No program corresponds to this process, as it is part of the kernel and is a system process.
- PID 1 is the *init* process.
  It is invoked by the kernel at the end of the bootstrap procedure (stored in */sbin/init*).
  It is responsible for bringing up the system after bootstrap of the kernel and it never dies.
- PID 2 is the *pagedaemon* in many UNIX implementations.
  Responsible for supporting the paging of the virtual memory system.
- In addition to PID, there are other identifiers that are used for every process.
  The following table gives the functions that return these identifiers.

Function	Return Values
pid_t getpid(void);	Process ID of calling process
pid_t getppid(void);	Parent process ID of calling process
uid_t getuid(void);	Real user ID of calling process
uid_t geteuid(void);	Effective user ID of calling process
gid_t getgid(void);	Real group ID of calling process
gid_t getegid(void);	Effective group ID of calling process

(XXXVII) fork Function

- Existing process can create a new process using the *fork()* function.

```
#include<unistd.h>

pid_t fork(void);
```

- It returns a value of 0 in child process and PID of child process in parent process if successful, -1 on error.
- The new process created by *fork* is called a *child process*.
- The *fork* function is called once but it returns two values – value for child process and PID of child process in parent process.
- The reason why the child's PID is returned to the parent is because a process can have more than one child process, and there is no function that allows a process to obtain the PIDs of its children.
- The value of 0 is returned to child process because a process can have only a single parent, and the child process can always call *getppid* to obtain the PID of its parent.
- PID 0 is reserved for use by the kernel, so it is not possible to be the PID of a child.

- Both child and parent continue executing after *fork* function call.
- Child is a copy of the parent, i.e., it gets a copy of the parent's data space, heap and stack segments.
- Only the text segment is shared between parent and child.
- *Copy-on-write (COW)* refers to a concept applicable to shared regions between parent and child, which are initially read-only.
- If either process tries to modify, then only that part of memory will be copied as a page in a virtual memory system.
- It is not known if child executes before parent or vice versa; it completely depends on the scheduling algorithm used by kernel.
- If synchronization between parent and child is necessary, interprocess communication (IPC) is required.
- 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.

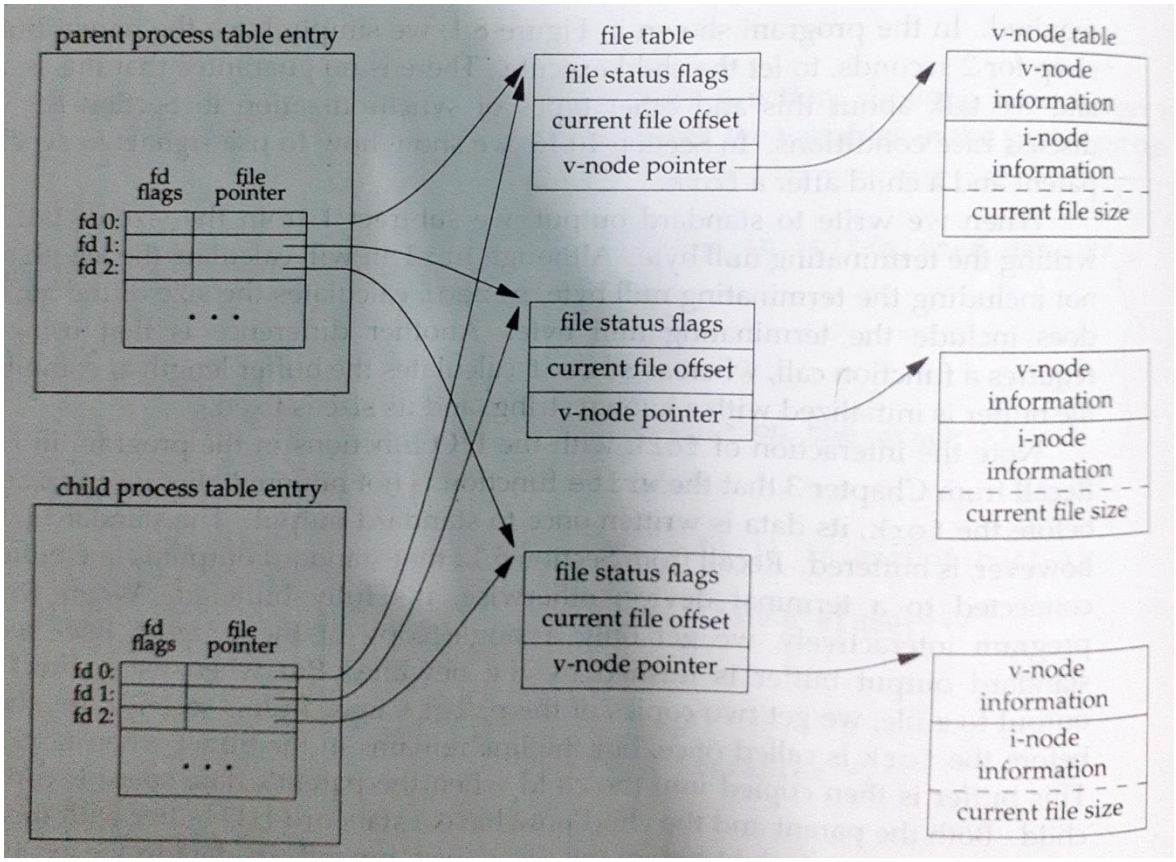
```
#include "apue.h"

int glob = 6;
char buf[] = "a write to stdout\n";

int main(void){
        int var;
        pid_t pid;
        var = 88;
        if(write(STDOUT_FILENO, buf, sizeof(buf)-1)!=sizeof(buf)-1)
                err_sys("write error!");
        printf("before fork\n");
        if((pid = fork()) < 0)
                err_sys("fork error!");
        else if(pid == 0){
                glob++;
                var++;
        }
        else
                sleep(2);
        printf("pid=%d, lob=%d, var=%d\n", getpid(), glob, var);
        exit(0);
}
```

- The parent puts itself to sleep for 2 seconds, to let the child execute.
- When we write to standard output, we subtract 1 from the size of *buf* to avoid writing the terminating null byte.

- Although *strlen* will calculate the length of a string not including the terminating null byte, *sizeof* calculates the size of the buffer, which does include the terminating null byte.
- Another difference is that using *strlen* requires a function call, whereas *sizeof* calculates the buffer length at compile time, as the buffer is initialized with a known string, and its size is fixed.
- *write* is called before the *fork*, thus it is not buffered and its data is written once to standard output.
  The standard I/O library, however, is 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*** – all file descriptors that are open in the parent are duplicated in the child.
- Parent and child share a file table entry for every open descriptor.
- Parent and child should share the same file offset.
- *Example* - process forks a child, then waits for child to complete.
- Assumption – both processes write to standard output as part of normal processing.
- If standard output for parent is redirected, child should update the parent's file offset when it completes.
- Child can write to standard output when parent is waiting for it and then the parent can continue writing to standard output after the child has completed execution – parent's output will be appended to child's output.

- 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 below.



- Two normal cases to handle file descriptors after *fork* -
  - Parent waits for child to complete.
    Parent does not need to do anything with its file descriptors.
    When child terminates, any of the shared descriptors will update file offsets accordingly.
  - Parent and child execute independently.
    Parent closes the descriptors that it does not need, child does the same.
    Neither interferes with the other's open descriptors.
- The properties of parent process that are inherited by the child process are –
  - Real user ID, real group ID, effective user ID, effective group ID
  - Supplementary group IDs
  - Process group ID
  - Session ID
  - Controlling terminal
  - *set-user-ID* and *set-group-ID* flags
  - Current working directory
  - Root directory
  - File mode creation mask

- Signal mask and dispositions
- *close-on-exec* flag for any open file descriptors
- Environment
- Attached memory segments
- Memory mappings
- Resource limits

- The differences between parent process and child process are –
  - Return value from *fork*
  - PIDs
  - Parent PIDs – parent PID of child is the parent; parent PID of parent does not change
  - Child's *tms_utime, tms_stime, tms_cutime, tms_cstime* values are set to 0
  - File locks set by parent are not inherited by child
  - Pending alarms are cleared for the child
  - Set of pending signals for the child is set to the empty set
- There are a couple of reasons for *fork* to fail –
  - If too many processes are already in the system, *fork* usually fails and this indicates that something else is wrong in the system.
  - If total number of process for real user ID exceeds the system limit, then the *fork* function usually fails.

  The constant CHILD_MAX describes the maximum number of simultaneous processes per real user ID.
- There are a couple of uses for *fork* -
  - When a process wants to duplicate itself so that parent and child can execute different sections of the code at the same time, *fork* can be used.

    This is common for network servers.

    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.
  - When a process wants to execute a different program, *fork* can be used.

    This is common for shells.

    In this case, the child does an *exec* right after it returns from the *fork*.
- Some operating systems combine the operations from *fork* followed by an *exec* into a single operation called a *spawn*.

  The UNIX System separates the two, as there are numerous cases where it is useful to *fork* without doing an *exec*.

  Also, separating the two allows the child to change the per-process attributes between the *fork* and the *exec*, such as I/O redirection, user ID, signal disposition.

(XXXVIII) vfork Function

- *vfork* has same calling sequence and return values as *fork*, but semantics differ.
- *vfork* is intended to create a new process when the purpose of the new process is to *exec* a new program.
- It creates a new process just like *fork*, without copying the address space of the parent into the child.
  Child won't reference that address space, it simply calls *exec* or *exit* right after *vfork*.
- While child is running and until it calls either *exec* or *exit*, child runs in address space of parent.
- This optimization provides efficiency gain on some paged virtual memory implementations.
- Another difference between *fork* and *vfork* 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.
- There is a high possibility of deadlock if the child depends on further actions of the parent before calling either of these two functions.
- The program below is a modified version of the program used for *fork*.

```
#include "apue.h"

int glob = 6;

int main(void){
        int var;
        pid_t pid;
        var = 88;
        printf("before vfork\n");
        if((pid=vfork()) < 0)
                err_sys("vfork error!");
        else if(pid == 0){
                glob++;
                var++;
                _exit(0);
        }
        //parent process continues
        printf("pid=%d, glob=%d, var=%d\n", getpid(), glob, var);
        exit(0);
}
```

- The call to *fork* has been replaced with *vfork* and the *write* to standard output has been removed.
- Also, the parent does not explicitly call *sleep*, as it is guaranteed that the parent is put to sleep by the kernel until the child calls either *exec* or *exit*.
- 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 doesn't surprise us. This behavior, however, differs from fork.
- *_exit* does not perform any flushing of standard I/O buffers.
  If *exit* is called instead, the results are indeterminate.
- Depending on the implementation of the standard I/O library, there might be no difference in the output, or the output from the parent's *printf* may disappear.
- If the child calls *exit*, the implementation flushes the standard I/O streams.
- If this is the only action taken by the library, then no difference can be observed with the output generated if the child called *_exit*.
- If the implementation also closes the standard I/O streams, however, the memory representing the FILE object for the standard output will be cleared out.
- When the parent resumes and calls *printf*, no output will appear and *printf* will return -1 because the child is borrowing the parent's address space.
- The parent's STDOUT_FILENO is still valid, as the child gets a copy of the parent's file descriptor array.
- Most modern implementations of exit will not bother to close the streams.
- The kernel will close all the file descriptors open in the process because the process is about to exit.
- Closing them in the library simply adds overhead without any benefit.

(XXXIX) exit Functions

- There are five ways of normal termination of a process –
  - Executing *return* from the *main* function.
    This is equivalent to calling *exit*.
  - Calling the *exit* function.
    This is defined by ISO C and includes calling of all exit handlers that have been registered by calling *atexit* and closing all standard I/O streams.
  - Calling *_exit* or *_Exit* function.
    ISO C defines *_Exit* to terminate a process without running exit handlers or signal handlers.
    In the UNIX System implementation, both are synonymous and do not flush standard I/O streams.
    _exit is called by *exit* and handles UNIX system-specific details, and is specified by POSIX.1.
  - Executing *return* from start routine of the last thread in the process.
    Return value of thread is not used as the return value of the process.
    When last thread returns from its start routine, process exits with termination status of 0.
  - Calling *pthread_exit* function from the last thread in the process.

Exit status of process is always 0, regardless of the argument passed to *pthread_exit*.

- There are three ways of abnormal termination of a process –
  - Calling *abort*.
    This generates the SIGABRT signal and interrupts the running process.
  - When the process receives certain signals.
    Signal can be generated by the process itself, or by some other process, or by the kernel.
    Examples - divide by 0, process referencing a memory location not within its address space.
  - Last thread responds to a cancellation request.
    One thread requests that another thread be canceled, and sometime later, the target thread terminates.
- Regardless of how a process terminates, same code in kernel is eventually executed.
- It closes all open descriptors for the process and releases memory that it was using.
- Terminating process should notify the parent how it terminated –
  - For the 3 exit functions, process passes exit status as argument to the function.
  - For abnormal termination, kernel generates termination status to indicate the reason.
- Parent can obtain termination status from either *wait* or *waitpid* function.

- Exit status is different from termination status.
  - Exit status is the argument to one of 3 *exit* functions, or return value from *main*.
  - Termination status is when exit status is converted by the kernel when *_exit* is finally called.
- If child terminates normally, then parent can obtain the exit status of the child.

- *Case 1* – when parent terminates before child.
  - *init* process becomes parent process of the process whose parent terminated.
  - It is said that the process has been inherited by *init*.
  - When a process terminates, the kernel goes through all active processes to check if terminating process is parent of any process that still exists.
  - If so, parent PID of surviving process changed to 1 (PID of *init*); this guarantees that every process has a parent.
- *Case 2* – when child terminates before parent.
  - If child completely disappears, parent will not be able to fetch termination status.
  - Kernel keeps information for every terminating process, so parent can fetch this information when it calls *wait* or *waitpid*.
  - This information consists of PID, termination status, amount of CPU time taken by the process.
  - Kernel discards all memory used by the process and closes its open files.

- A process that has terminated but whose parent has not yet waited for it is known as a *zombie* process.
- ***ps*** command prints state of zombie process as ***Z***.
- Many child processes become zombies; the parent has to wait and fetch termination status of each zombie process in this case.

- *Case 3* – when process inherited by *init* terminates.
  - *init* calls one of the *wait* functions and fetches the termination status; the process does not become a zombie.
  - *init* prevents the system from being clogged by zombies.
  - If a process is *init*'s child, it is a process that is generated by *init* directly, or process whose parent has been terminated and has been subsequently inherited by *init*.

(XL) 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.
- Termination of a child process is asynchronous event – it can happen at any time while the parent is running.
- *SIGCHLD* is the asynchronous notification from kernel to parent.
- Parent can ignore it or use a *signal handler*.
- The default action is to ignore the signal.
- Process that calls *wait* or *waitpid* can do the following –
  - Block, if all its children are running
  - Return immediately with termination status of a child, if child has terminated and waiting for termination status to be fetched.
  - Return immediately with an error, if it does not have any child processes.
- If process calls *wait* because it received *SIGCHLD* signal, then *wait* will return immediately.
- If process calls *wait* at random point in time, it can block.
- Prototypes of *wait* and *waitpid* functions are given below –

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

pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

- Return values are the PID if successful, -1 on error.
- The differences between *wait* and *waitpid* functions are as follows –
  - *wait* can block the caller until a child process terminates.
  waitpid has an option that prevents it from blocking.
  - *waitpid* does not wait for the child that terminates first. It has several options that

control which process it waits for.
wait does not have these options and it waits for all child processes that terminate.

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

- *statloc* is a pointer to an integer.
- If *statloc* is not null pointer, termination status of terminated process is stored in the location pointed to by *statloc*.
- If *statloc* is null pointer, nothing will be stored.
- Earlier, termination status had multiple bits – exit status (normal return), signal number (abnormal return), core file generation.
- Currently, in POSIX.1 implementation, termination status can be referred to by using 4 mutually exclusive macros defined in *<sys/wait.h>*
- These macros tell us how the process terminated and all begin with *WIF*.

Macro	**Description**
WIFEXITED(status)	True if status was returned for a child that terminated normally. Execute **WEXITSTATUS(status)** to fetch low-order 8 bits of argument that child passed to either of the 3 exit functions.
WIFSIGNALED(status)	True if status was returned for a child that terminated abnormally, by receipt of a signal that was not caught. Execute **WTERMSIG(status)** to fetch signal number that caused termination. In some implementations, **WCOREDUMP(status)** returns true if core file of terminated process was generated.
WIFSTOPPED(status)	True if status was returned for a child that is currently stopped. Execute **WSTOPSIG(status)** to fetch signal number that caused the child to stop.
WIFCONTINUED(status)	True if status was returned for a child that has been continued after a job control stop.

- The function *pr_exit* uses these macros to print a description of the termination status. This function also handles WCOREDUMP macro, if it is defined.

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

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

- The program below calls the *pr_exit* function, demonstrating the various values for the termination status.

  Unfortunately, there is no portable way to map the signal numbers from WTERMSIG into descriptive names.

  The <signal.h> header has to be referred to verify that SIGABRT has a value of 6 and that SIGFPE has a value of 8.

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

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

        if((pid = fork()) < 0)
                err_sys("Fork error");
        else if(pid == 0) //child
                exit(7);

        if(wait(&status) != pid) //wait for child
                err_sys("Wait error");
        pr_exit(status); //and print its status

        if((pid = fork()) < 0)
                err_sys("Fork error");
        else if(pid == 0) //child
                abort();  //generates SIGABRT
```

```
        if(wait(&status) != pid) //wait for child
                err_sys("Wait error");
        pr_exit(status); //and print its status

        if((pid = fork()) < 0)
                err_sys("Fork error");
        else if(pid == 0)    //child
                status /= 0; //generates SIGFPE

        if(wait(&status) != pid) //wait for child
                err_sys("Wait error");
        pr_exit(status); //and print its status

        exit(0);
}
```

- In older UNIX implementations, if a specific process had to be fetched for wait, the procedure was tedious and given as follows: call *wait*, compare the PID with desired PID, save PID and termination status and call *wait* again.
  In the next *wait*, scrape through list of already terminated process, call *wait* again.
- This was improved significantly in the current implementation, wherein *waitpid* waits for a specific process to terminate, assuming the PID is known.

***pid* value**	**Interpretation**
***pid* == -1**	Waits for any child process. *waitpid* becomes equivalent to *wait*.
***pid* > 0**	Waits for the child whose PID equals *pid*.
***pid* == 0**	Waits for any child process whose process group ID equals that of the calling process.
***pid* < -1**	Waits for any child process whose process group ID equals the absolute value of *pid*.

- Error occurs for *wait* if calling process has no children, or is interrupted by a signal.
- Error occurs for *waitpid* if calling process has no children, or is interrupted by a signal, or specified process/process group does not exist, or is not child of calling process.
- *options* argument allows for more control in the operation of *waitpid*.
- *options* is either 0 or constructed from bitwise-OR of the *options* constants.

Constant	**Description**
WCONTINUED	Status of any child specified by *pid* that has been continued after being stopped, but whose status has not yet been reported, is returned (if

	implementation supports job control).
WNOHANG	*waitpid* will not block if a child specified by *pid* is not immediately available. Return value is 0.
WUNTRACED	Status of any child specified by *pid* that has stopped, and whose status has not been reported since it stopped, is returned (if implementation supports job control). **WIFSTOPPED** macro determines whether return value corresponds to a stopped child process.

- The features of *waitpid* which are not provided by *wait* are given as follows –
  - *waitpid* allows us to wait for one particular process, whereas *wait* returns the status of any terminated child.
  - *waitpid* provides non-blocking version of *wait*.
  This is useful when child's status must be fetched, but blocking is not necessary.
  - *waitpid* provides support for job control with **WCONTINUED** and **WUNTRACED** options.
- The program below is used to avoid zombie processes by calling *fork* twice.

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

int main(void){
        pid_t pid;
        if((pid=fork()) < 0)
                err_sys("Fork error");
        else if(pid == 0){ //first child
                if((pid=fork()) < 0)
                        err_sys("Fork error");
                else if(pid > 0)
                        exit(0); //parent from first fork
                sleep(2);
                printf("Second child, parent PID = %d\n", getppid());
                exit(0);
        }
        if(waitpid(pid, NULL, 0) != pid) //wait for first child
                err_sys("waitpid error");
        exit(0);
}
```

The process forks a child but it does not wait for the child to complete and the child does not become a zombie until it is manually terminated.

sleep is called in the second child to ensure that the first child terminates before printing the parent PID.

After a *fork*, either the parent or child can continue executing; the process that resumes execution first is not known.

If the second child is not put to sleep, and if it resumed execution after the *fork* before

its parent, the parent PID that it printed would be that of its parent, not PID 1.
The shell prints its prompt when the original process terminates, which is before the second child prints its parent PID.

(XLI) wait3 and wait4 Functions

- These functions descend from BSD branch of UNIX System implementation.
- These functions contain an additional argument (*rusage*) that allows kernel to return a summary of resources used by terminated process and all its child processes.

```
#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);
```

- The resource information obtained – amount of user CPU time, amount of system CPU time, number of page faults, number of signals received.

(XLII) Race Conditions

- Race condition occurs when multiple processes are accessing shared data and final outcome depends on the order in which these processes run.
- *fork* is prime example for occurrence of race conditions, if the logic after *fork* explicitly or implicitly depends on whether parent runs first or child.
  The user cannot predict which runs first.
  Even if first running process is known, what happens after the *fork* depends on system load and kernel's scheduling algorithm.
- If a process wants to wait for child to terminate, then it must call one of the *wait* functions.
- If a process wants to wait for parent to terminate, then *polling* loop must be used.
- Polling refers to the situation when the caller is awakened every second to test if process is terminated.
  It wastes CPU time because of continuous interruptions.
- To avoid race conditions and to avoid polling, signalling between multiple processes and interprocess communication (IPC) can be used.
- Parent and child scenario after *fork* – parent and child communicate with each other about the operations being performed and synchronize accordingly.

```c
#include "apue.h"

TELL_WAIT(); //set things up for TELL and WAIT

if((pid=fork()) < 0)
        err_sys("fork error");
else if(pid == 0){ //child
        //child operations
        TELL_PARENT(getppid()); //specify end
        WAIT_PARENT(); //wait for parent
        //child operations continue
        exit(0);
}
//parent operations
TELL_CHILD(pid); //specify end
WAIT_CHILD(); //wait for child
//parent operations continue
exit(0);
```

TELL and WAIT routines can also be implemented using signals and pipes.

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

  The program contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.

```c
#include "apue.h"

static void charatatime(char *);

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

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

- To avoid the race condition, the program is modified to include the TELL and WAIT functions, as follows.

```
#include "apue.h"

static void charatatime(char *);

int main(void){
        pid_t pid;
        TELL_WAIT();
        if((pid=fork()) < 0)
                err_sys("fork error");
        else if(pid == 0){
                WAIT_PARENT(); //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);
}
```

(XLIII) exec Functions

- When a process calls one of the *exec* functions, that process is completely replaced by the new program and it starts executing at its *main* function.
- PID does not change across an *exec* because new process is not created.
- *exec* just replaces the current process (text, data, stack and heap segments) with a brand new program from disk.
- Six different *exec* functions are present; these are collectively known as "the *exec* function" and any one of the six can be used.
- Prototypes of the 6 *exec* functions are given below –

```
#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[]);
```

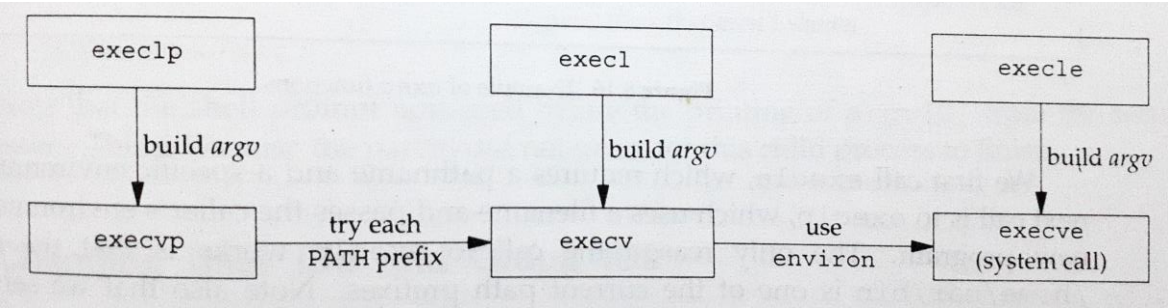
- Return values for all 6 functions – no return on success, -1 on error.
- The main differences between the *exec* functions are given below –
  - *execlp* and *execvp* take filenames as arguments, remaining take pathnames.
    If filename contains a slash, it is taken as pathname.
    Otherwise, executable file is searched in the directories defined in **PATH** environment variable.
  - *execl, execle, execlp* require arguments to be passed as a list of separate arguments.
    execv, execve, execvp require arguments to be passed as an array of pointers.
  - *execle, execve* allow environment list to be passed as a pointer to an array of pointers which consist of the environment strings.
    Remaining 4 use *environ* global variable to copy existing environment to new program.

- Remembering the arguments to the *exec* functions –
  - *p* – function takes a filename argument and uses PATH environment variable to find the executable file (*execlp, execvp*)
  - *l* – function takes a list of arguments (*execl, execle, execlp*)
  - *v* – function takes a vector (argv[]) of arguments (*execv, execve, execvp*)
  - *e* – function takes *envp[]* array instead of using current environment (*execle, execve*)
- The limit on total size of argument list and environment list is given by ARG_MAX. This value must be at least 4096 bytes on a POSIX.1 system.

- PID does not change after an *exec*, but new program inherits additional properties from calling process –
  - PID and Parent PID
  - Real user ID and real group ID
  - Supplementary group IDs
  - Process group ID
  - Session ID
  - Controlling terminal
  - Time left until alarm clock
  - Current working directory
  - Root directory
  - File mode creation mask
  - File locks
  - Process signal mask
  - Pending signals
  - Resource limits
  - Values for *tms_utime, tms_stime, tms_cutime, tms_cstime*

- Handling of open files depends on the value of close-on-exec flag for each descriptor.
- If it is set, descriptor is closed across an *exec*.
  Otherwise, it is left open across an *exec*.
- By default, the descriptor is left open across an *exec* unless *close-on-exec* is specifically set by *fcntl*.
- POSIX.1 requires that open directory streams be closed across an *exec*.
  This is normally done by *opendir* calling *fcntl* to set the close-on-exec flag for the descriptor corresponding to the open directory stream.

- Real user ID and real group ID remain the same across *exec*, but effective IDs can change depending on the status of set-user-ID and set-group-ID bits for the program.
- If set-user-ID bit is set for new program, effective user ID becomes real user ID of the program file.
  Otherwise, effective user ID is not changed.
  The same ID handling mechanism is applicable for group IDs as well.
- In most UNIX system implementations, only *execve* is a system call within the kernel.
  The remaining 5 are just library functions that eventually invoke this system call.
- The relationship between the 6 *exec* functions is given in the figure below.
  The library functions *execlp* and *execvp* process the PATH, looking for the first path prefix that contains an executable file named *filename*.



- The program below demonstrates the *exec* functions.
  execle is called first, 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 call to *execlp* works because the directory */home/sar/bin* is one of the current path prefixes.
  The first argument *argv[0]* in the new program is set to be the filename component of the pathname.

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

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

int main(void){
        pid_t pid;
        if((pid=fork()) < 0)
                err_sys("fork error");
        else if(pid == 0) //specify pathname, environment
                if(execle("/home/sar/bin/echoall", "echoall",
                        "myarg1", "MY ARG2", (char *)0, env_init)<0)
                        err_sys("execle error");
        if(waitpid(pid, NULL, 0) < 0)
                err_sys("wait error");
        if((pid=fork()) < 0)
                err_sys("fork error");
        else if(pid == 0)
                if(execlp("echoall","echoall", "only 1 arg"
                                (char *)0)<0)
                        err_sys("execlp error");
        exit(0);
}
```

- The program *echoall* that is executed twice in the above program is given below. It is a simple program that echoes all its command line arguments and its entire environment list.

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