CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Improvement Test

Sub:	Unix System Programming							Code:	10CS62
Date:	31/05/2017	Duration:	90 mins	Max Marks:	50	Sem:	VI	Branch:	ISE

Answer Any FIVE FULL Questions

		Marks	OBE CO	OBE RBT
1 (a)	What are signals? List any four signals along with brief explanation .Explain with a program how to set up a handler	[10]	CO4	L3
2 (a)	What is daemon process? Discuss its characteristics and coding rules	[10]	CO5	L2
3 (a)	Discuss UNIX and POSIX standards in detail. Write a C/C++ program to check the following limits a. Clock ticks b. Max number of child process c. Max path length d. Max file name e. Max characters in file	[10]	CO2	L2
4 (a)	Explain briefly kill( ) API and alarm( ) API	[10]	CO4	L3
5 (a)	What do you mean by fork( ) and vfork( ) functions ? Explain both the functions with example programs (write separate programs)	[10]	CO4	L2
6 (a)	What are file? Discuss various file types in UNIX in detail with proper example.	[10]	CO2	L2
7 (a)	Explain the major difference between ANSI ‘C’ and K & R ‘C’ with example in detail.	[10]	CO1	L2
8 (a)	Explain the memory layout of ‘C’ Program	[5]	CO4	L2
8(b)	Write the prototypes of system library calls available to manipulate shared memory	[5]	CO6	L2

**********All the best**********

#132, AECS Layout, IT Park Road, Kundalahalli, Bangalore – 560 037, T:+9180 28524466 / 77

CMR
INSTITUTE OF
TECHNOLOGY

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

Scheme and solution for Improvement Test-May 2017

Q. 1 a) What are signals? List any four signals along with brief explanation .Explain with a program how to set up a handler-10

Definition signal -2M

Signals are software interrupts. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

Any four signal with brief explanation-4M

Name	Description	Default action
SIGABRT	abnormal termination (abort)	terminate+core
SIGALRM	timer expired (alarm)	terminate
SIGBUS	hardware fault	terminate+core
SIGCANCEL	threads library internal use	ignore
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGEMT	hardware fault	terminate+core
SIGFPE	arithmetic exception	terminate+core
SIGFREEZE	checkpoint freeze	ignore
SIGHUP	hangup	terminate
SIGILL	illegal instruction	terminate+core
SIGINFO	status request from keyboard	ignore
SIGINT	terminal interrupt character	terminate

Program-4M

The following example attempts to catch the SIGTERM signal, ignores the SIGINT signal, and accepts the default action of the SIGSEGV signal. The pause API suspends the calling process until it is interrupted by a signal and the corresponding signal handler does a return:

```
#include<iostream.h>
#include<signal.h>
/*signal handler function*/
void catch_sig(int sig_num)
{
      signal (sig_num,catch_sig);

        cout<<"catch_sig:"<<sig_num<<endl;
 }

 /*main function*/
 int main()
 {
        signal(SIGTERM,catch_sig);
        signal(SIGINT,SIG_IGN);
        signal(SIGSEGV,SIG_DFL);
        pause( );                /*wait for a signal interruption*/
 }
```

Q. 2a) What is daemon process? Discuss its characteristics and coding rules

Characteristics-4M

The characteristics of daemons are:

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don't have controlling terminal.
- Daemons are session and group leaders.

Coding Rules-Program 3M

Explanation -3M

CODING RULES

- **Call `umask` to set the file mode creation mask to 0**. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions.
- **Call `fork` and have the parent `exit`**. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader.
- **Call `setsid` to create a new session**. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.
- **Change the current working directory to the root directory**. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.
- **Unneeded file descriptors should be closed.** This prevents the daemon from holding open any descriptors that it may have inherited from its parent.
- **Some daemons open file descriptors 0, 1, and 2 to `/dev/null` so that any library routines that try to read from standard input or write to standard output or standard error will have no effect**. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed; nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

Example Program:

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

int daemon_initialise( )
 {
       pid_t pid;
       if (( pid = for() ) < 0)
              return -1;
       else if ( pid != 0)
              exit(0);         /* parent exits */

       /* child continues */
       setsid( );
       chdir("/");
       umask(0);
       return 0;
}
```

Q.3a) Discuss UNIX and POSIX standards in detail.
Write a C/C++ program to check the following limits

a. Clock ticks
b. Max number of child process
c. Max path length
d. Max file name
e. Max characters in file

POSIX standards-5M

The POSIX standards

- POSIX or "Portable Operating System Interface" is the name of a family of related standards specified by the IEEE to define the application-programming interface (API), along with shell and utilities interface for the software compatible with variants of the UNIX operating system.
- Because many versions of UNIX exist today and each of them provides its own set of API functions, it is difficult for system developers to create applications that can be easily ported to different versions of UNIX.
- Some of the subgroups of POSIX are POSIX.1, POSIX.1b & POSIX.1c are concerned with the development of set of standards for system developers.
- **POSIX.1**
  - This committee proposes a standard for a base operating system API; this standard specifies APIs for the manipulating of files and processes.
  - It is formally known as IEEE standard 1003.1-1990 and it was also adopted by the ISO as the international standard ISO/IEC 9945:1:1990.
- **POSIX.1b**
  - This committee proposes a set of standard APIs for a real time OS interface; these include IPC (inter-process communication).
  - This standard is formally known as IEEE standard 1003.4-1993.
- **POSIX.1c**
  - This standard specifies multi-threaded programming interface. This is the newest POSIX standard.
  - These standards are proposed for a generic OS that is not necessarily be UNIX system.
  - E.g.: VMS from Digital Equipment Corporation, OS/2 from IBM, & Windows NT from Microsoft Corporation are POSIX-compliant, yet they are not UNIX systems.
  - To ensure a user program conforms to POSIX.1 standard, the user should either define the manifested constant _POSIX_SOURCE at the beginning of each source module of the program (before inclusion of any header) as;

    ```
    #define _POSIX_SOURCE
    ```

    Or specify the -D_POSIX_SOURCE option to a C++ compiler (CC) in a compilation;

    ```
    %       CC -D_POSIX_SOURCE *.C
    ```
  - POSIX.1b defines different manifested constant to check conformance of user program to that standard. The new macro is _POSIX_C_SOURCE and its value indicates POSIX version to which a user program conforms. Its value can be:

_POSIX_C_SOURCE VALUES	MEANING
198808L	First version of POSIX.1 compliance
199009L	Second version of POSIX.1 compliance
199309L	POSIX.1 and POSIX.1b compliance

- _POSIX_C_SOURCE may be used in place of _POSIX_SOURCE. However, some systems that support POSIX.1 only may not accept the _POSIX_C_SOURCE definition.
- There is also a _POSIX_VERSION constant defined in <unistd.h> header. It contains the POSIX version to which the system conforms.

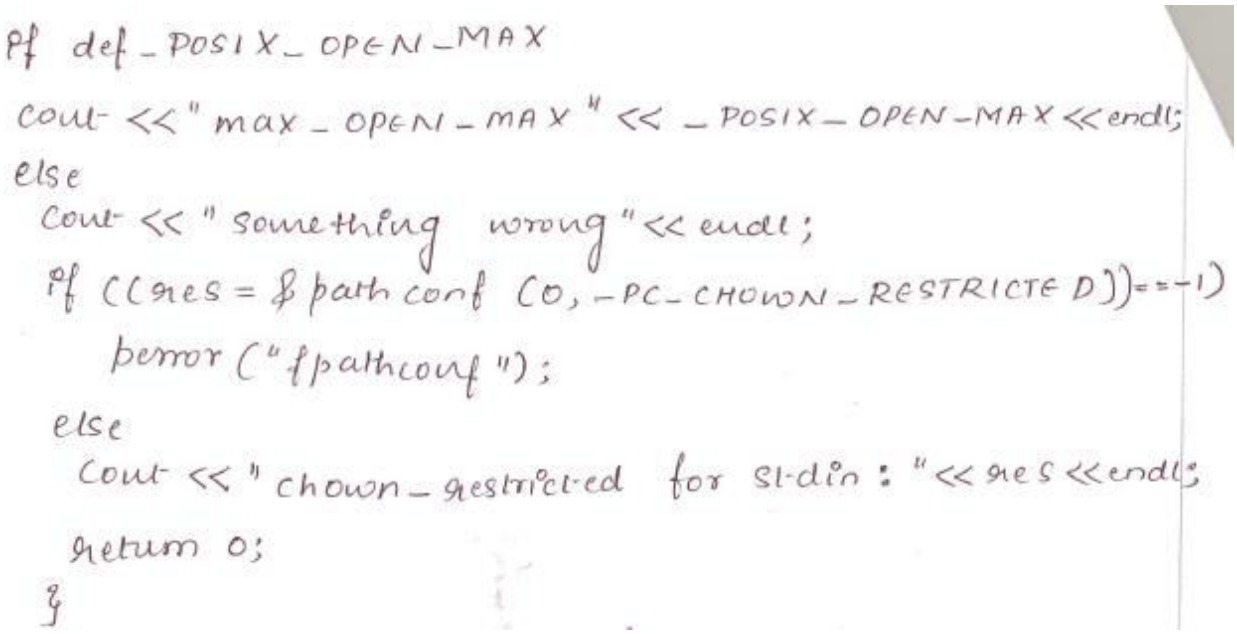
Although POSIX was developed on UNIX, a POSIX complaint system is not necessarily a UNIX system. A few UNIX conventions have different meanings according to the POSIX standards. Most C and C++ header files are stored under the /usr/include directory in any UNIX system and each of them is referenced by

```
#include<header-file-name>
```

This method is adopted in POSIX. There need not be a physical file of that name existing on a POSIX conforming system.

Program-5M

```
#define  _POSIX_SOURCE
#define  _POSIX_C_SOURCE  199309L
#include <stdio.h>
#include <iostream.h>
#include <unistd.h>
int main()
{
  int res;
  if ((res ==sysconf(_SC_CLK_TCK)) == -1)
  perror ("sysconf");
  else
     cout << "CLK_TCK" << res << endl;
   if (res == sysconf(_SC_CHILD_MAX)) == -1)
       perror ("sysconf");
    else
      cout<< "CHILD_MAX"<<res<<endl;
     if ((re == pathconf ("/", _PC_PATH_MAX)) == -1)
       perror ("pathconf");
     else
       cout << " PATH_MAX " << (res+1)<< endl;
```



```
#if def _POSIX_OPEN_MAX
cout <<" max_OPEN_MAX " << _POSIX_OPEN_MAX <<endl;
else
 cout <<" something wrong" <<endl;
 if ((res = fpathconf (0, _PC_CHOWN_RESTRICTED))==-1)
    perror ("fpathconf");
 else
  cout <<" chown_restricted for stdin : " <<res <<endl;
  return 0;
}
```

Q. 4a) Explain briefly kill( ) API and alarm( ) API

Kill API-Prototype-2M

Explanation with example-3M

A process can send a signal to a related process via the kill API. This is a simple means of inter-process communication or control. The function prototype of the API is:

```
#include<signal.h>
int kill(pid_t pid, int signal_num);
```

Returns: 0 on success, -1 on failure.

The following program illustrates the implementation of the UNIX kill command using the kill API:

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

int main(int argc,char** argv)
{
        int pid, sig = SIGTERM;
        if(argc==3)
        {
                if(sscanf(argv[1],"%d",&sig)!=1)
                {
                        cerr<<"invalid number:" << argv[1] << endl;
                        return -1;
                }
                argv++,argc--;
        }
        while(--argc>0)
        if(sscanf(*++argv, "%d", &pid)==1)
        {
                if(kill(pid,sig)==-1)
                        perror("kill");
        }
        else
                cerr<<"invalid pid:" << argv[0] <<endl;
        return 0;
}
```

The UNIX kill command invocation syntax is:

Kill [-<signal_num>] <pid>.....

Where signal_num can be an integer number or the symbolic name of a signal. <pid> is process ID.

Alarm API( )-Prototype-2M

Explanation with example – 3M

The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds. The function prototype of the API is:

```
#include<signal.h>
Unsigned int alarm(unsigned int time_interval);
```

Returns: 0 or number of seconds until previously set alarm

The alarm API can be used to implement the sleep API:

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

void wakeup( )
{ ; }

unsigned int sleep (unsigned int timer )
{
       struct sigaction action;
       action.sa_handler=wakeup;
       action.sa_flags=0;
       sigemptyset(&action.sa_mask);
       if(sigaction(SIGALARM,&action,0)==-1)
       {
              perror("sigaction");
              return -1;
       }
       (void) alarm (timer);
       (void) pause( );
       return 0;
 }
```

Q. 5a) What do you mean by fork( ) and vfork( ) functions ? Explain both the functions with example programs (write separate programs)

Fork() prototype +explanation -2M

Program-3M

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

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

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

- The new process created by `fork` is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason `fork` returns 0 to the child is that a process can have only a single parent, and the child can always call `getppid` to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)
- Both the child and the parent continue executing with the instruction that follows the call to `fork`.
- The child is a copy of the parent.
- For example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; the parent and the child do not share these portions of memory.

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

#include<unistd.h>

int a=10;

int main()

{

        Pid_t pid; int b=20;

        Pid=fork();

        If(pid==0)

        {           a++;b++;}

        else { sleep() ;}

        printf("a= %d b= %d  pid= %d ppid= %d",&a,&b, getpid(),getppid());

}
```

Output:

a= 10 b=20 pid=100 ppid=200

a=11 b=21 pid=230 ppid=100

vfork( ) prototype +explanation=2M

Program+explanation-3M

vfork FUNCTION

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

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

Example of `vfork` function

```
#include "apue.h"
int     glob = 6;       /* external variable in initialized data */

int main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }
    /*
     * Parent continues here.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Output:

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

Q. 6a) What are file? Discuss various file types in UNIX in detail with proper example.

File Definition-1M

Files are the building blocks of any operating system. When you execute a command in UNIX, the UNIX kernel fetches the corresponding executable file from a file system, loads its instruction text to memory, and creates a process to execute the command on your behalf. In the course of execution, a process may read from or write to files. All these operations involve files. Thus, the design of an operating system always begins with an efficient file management system.

Regular file -1M

Each file type + explanation-4*2M=8M

File Types

A file in a UNIX or POSIX system may be one of the following types:

- regular file
- directory file
- FIFO file
- Character device file
- Block device file

- **Regular file**
  - A regular file may be either a text file or a binary file
  - These files may be read or written to by users with the appropriate access permission
  - Regular files may be created, browsed through and modified by various means such as text editors or compilers, and they can be removed by specific system commands
- **Directory file**
  - It is like a folder that contains other files, including sub-directory files.
  - It provides a means for users to organise their files into some hierarchical structure based on file relationship or uses.
  - Ex: **/bin** directory contains all system executable programs, such as **cat, rm, sort**
  - A directory may be created in UNIX by the **mkdir** command
    - Ex: `mkdir /usr/foo/xyz`
  - A directory may be removed via the **rmdir** command
    - Ex: `rmdir /usr/foo/xyz`
  - The content of directory may be displayed by the **ls** command

- **Device file**

Block device file	Character device file
It represents a physical device that transmits data a block at a time.	It represents a physical device that transmits data in a character-based manner.
Ex: hard disk drives and floppy disk drives	Ex: line printers, modems, and consoles

- A physical device may have both block and character device files representing it for different access methods.
- An application program may perform read and write operations on a device file and the OS will automatically invoke an appropriate device driver function to perform the actual data transfer between the physical device and the application
- An application program in turn may choose to transfer data by either a character-based(via character device file) or block-based(via block device file)
- A device file is created in UNIX via the **mknod** command
  - Ex: `mknod /dev/cdsk c 115 5`

Here , c - character device file
115 - major device number
5 - minor device number

❖ **FIFO file**

- It is a special pipe device file which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer.
- The size of the buffer is fixed to PIPE_BUF.
- Data in the buffer is accessed in a first-in-first-out manner.
- The buffer is allocated when the first process opens the FIFO file for read or write
- The buffer is discarded when all processes close their references (stream pointers) to the FIFO file.
- Data stored in a FIFO buffer is temporary.
- A FIFO file may be created via the **mkfifo** command.
  - The following command creates a FIFO file (if it does not exists)
    mkfifo /usr/prog/fifo_pipe
  - The following command creates a FIFO file (if it does not exists)
    mknod /usr/prog/fifo_pipe **p**
- FIFO files can be removed using **rm** command.

❖ **Symbolic link file**

- BSD UNIX & SV4 defines a symbolic link file.
- A symbolic link file contains a path name which references another file in either local or a remote file system.
- POSIX.1 does not support symbolic link file type
- A symbolic link may be created in UNIX via the **ln** command
- Ex: **ln -s /usr/divya/original /usr/raj/slink**
- It is possible to create a symbolic link to reference another symbolic link.
- **rm, mv** and **chmod** commands will operate only on the symbolic link arguments directly and not on the files that they reference.

Q. 7a) Explain the major difference between ANSI ‘C’ and K & R ‘C’ with example in detail.

Listing the major differences-2M

Each difference + explanation=2M*4=8M

The major differences between ANSI C and K&R C [Kernighan and Ritchie] are as follows:

- Function prototyping
- Support of the const and volatile data type qualifiers.
- Support wide characters and internationalization.
- Permit function pointers to be used without dereferencing.

Function prototyping

ANSI C adopts C++ function prototype technique where function definition and declaration include function names, arguments' data types, and return value data types. This enables ANSI C compilers to check for function calls in user programs that pass invalid number of arguments or incompatible arguments' data type.

These fix a major weakness of K&R C compilers: invalid function calls in user programs often pass compilation but cause programs to crash when they are executed.

Eg:
```
unsigned long foo(char * fmt, double data)
{
        /*body of foo*/
}
```

External declaration of this function foo is

```
unsigned long foo(char * fmt, double data);
```

eg:
```
int printf(const char* fmt,.....);
```

specify variable number of arguments

Support of the const and volatile data type qualifiers.

- The **const** keyword declares that some data cannot be changed.
  Eg: `int printf(const char* fmt,.....);`
  Declares a fmt argument that is of a const char * data type, meaning that the function printf cannot modify data in any character array that is passed as an actual argument value to fmt.
- **Volatile** keyword specifies that the values of some variables may change asynchronously, giving an hint to the compiler's optimization algorithm not to remove any "redundant" statements that involve "volatile" objects.

eg:
```
char get_io()
{
     volatile char* io_port = 0x7777;
     char ch = *io_port;        /*read first byte of data*/
     ch = *io_port;                    /*read second byte of data*/
}
```

If io_port variable is not declared to be volatile when the program is compiled, the compiler may eliminate second ch = *io_port statement, as it is considered redundant with respect to the previous statement.

Support wide characters and internationalisation

- ANSI C supports internationalisation by allowing C-program to use wide characters. Wide characters use more than one byte of storage per character.
- ANSI C defines the **setlocale** function, which allows users to specify the format of date, monetary and real number representations.
  For eg: most countries display the date in dd/mm/yyyy format whereas US displays it in mm/dd/yyyy format.
- Function prototype of setlocale function is:

```
#include<locale.h>
char setlocale (int category, const char* locale);
```

Permit function pointers without dereferencing

ANSI C specifies that a function pointer may be used like a function name. No referencing is needed when calling a function whose address is contained in the pointer.

For Example, the following statement given below defines a function pointer funptr, which contains the address of the function foo.

```
extern void foo(double xyz,const int *ptr);
void (*funptr)(double,const int *)=foo;
```

The function foo may be invoked by either directly calling foo or via the funptr.

```
foo(12.78,"Hello world");
funptr(12.78,"Hello world");
```

K&R C requires funptr be dereferenced to call foo.

```
(* funptr) (13.48,"Hello usp");
```

ANSI C also defines a set of C processor(cpp) symbols, which may be used in user programs. These symbols are assigned actual values at compilation time.

Q.8 a) Explain the memory layout of 'C' Program

Diagram-2M Explanation-3M

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

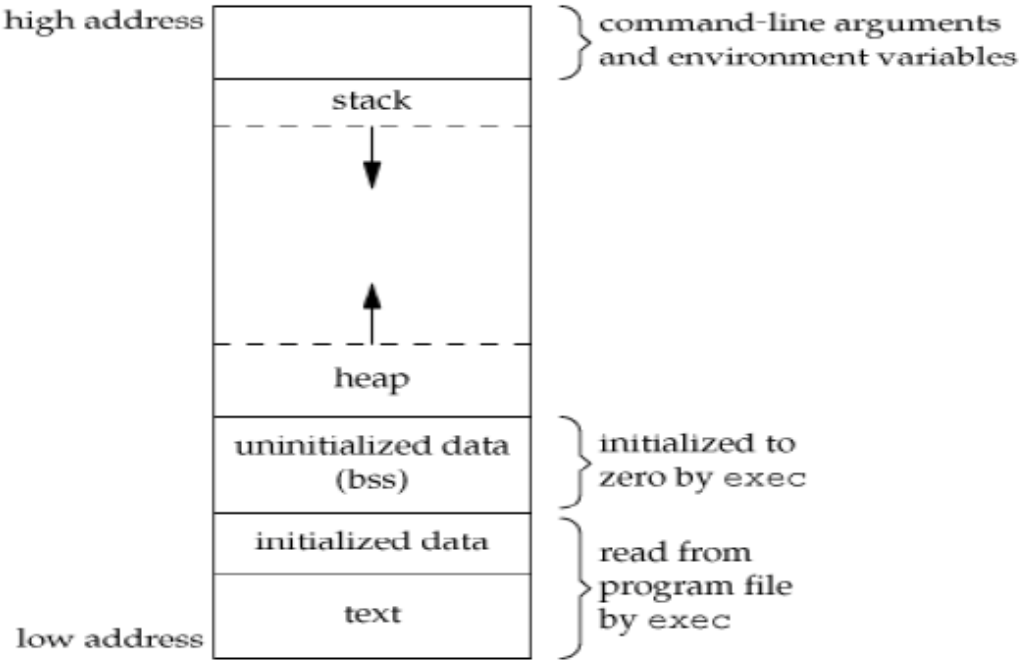
- **Text segment**, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent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

  ```
  int   maxcount = 99;
  ```

  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
- **Uninitialized data segment**, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration

  ```
  long  sum[1000];
  ```

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



Q.8 b) Write the prototypes of system library calls available to manipulate shared memory

Shared memory -1M

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server. The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access. (But as

Any two prototypes with explanation 2M*2=4M

The first function called is usually `shmget`, to obtain a shared memory identifier.

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

int shmget(key_t key, size_t size, int flag);
```

Returns: shared memory ID if OK, 1 on error

The `shmctl` function is the catchall for various shared memory operations.

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Returns: 0 if OK, 1 on error

Once a shared memory segment has been created, a process attaches it to its address space by calling `shmat`.

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

void *shmat(int shmid, const void *addr, int flag);
```

Returns: pointer to shared memory segment if OK, 1 on error