Scheme of Evaluation
Internal Assessment Test 3 – Jan 2022

Sub:	Unix Programming							**Code:**	18CS56
Date**:**	25/1/2022	Duration:	90mins	Max Marks:	50	**Sem:**	V	**Branch:**	ISE

Note: Answer Any five full questions.

Question #		**Description**	**Marks Distribution**		**Max Marks**
1	a)	Explain with neat diagram memory layout of C program. Diagram Explanation	 3M 4M	7M	10M
1	b)	Briefly discuss the different functions used for memory allocation Prototype with malloc calloc realloc	1M*3	3M	
2	a)	Compare and contrast 'fork' and 'vfork' system call with syntax and example. Any 6 differences	1M*6	6M	10M
2	b)	Write a program to explain setjmp and longjmp APIs . Main() Setjmp() Longjmp output	 1M 1M 1M 1M	4M	

3	a)	What are pipes? What are its limitations? Write a program to send a data from parent to child over a pipe Pipe-definition Limitation Program	1M 2M 3M		10M
	b)	What are Interpreter Files? Give the difference between Interpreter Files and Interpreter. Any three differences	1M*3	3M	
4	a)	Discuss popen and pclose functions with syntax and proper example. Prototype Usage example	2M 2M 2M	6M	10M
4	b)	What is FIFO? Explain the client server communication using FIFO Diagram Explanation	2M 2M	4M	
5	a)	Write a program to set up a signal hander for SIGINT and SIGALARM Signal handler for SIGINT Signal handler for SIGALARM Calling handler	2M 2M 2M	6M	10M
5	b)	What are signals? Mention different sources of signals with name and description. Signal Definition Any three signal names and description	1M 1M*3	4M	

6	a)	What is daemon process? Enlist its characteristics and coding rules. Daemon process Daemon process characteristics Coding rules	1M 2M 2M	5M	10M
	b)	What is error logging? With a neat block schematic discuss the error login facility in BSD Error logging Diagram Explanation	1M 2M 2M	5M	

Q. 1 a) Explain with neat diagram memory layout of C program.

MEMORY LAYOUT OF A C PROGRAM

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

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

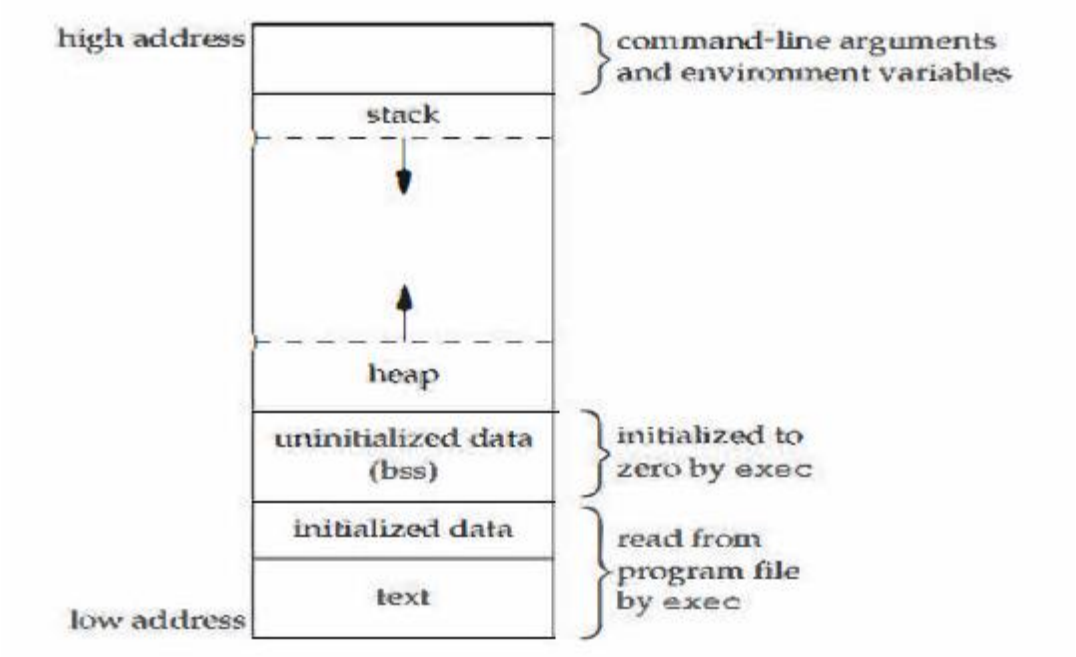
  int maxcount = 99;

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

long sum[1000];

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

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



Q.1 b) Briefly discuss the different functions used for memory allocation with syntax

MEMORY ALLOCATION

ISO C specifies three functions for memory allocation:

- malloc, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
- calloc, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
- realloc, which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size t newsize);
```

Q.2 a) Compare and contrast 'fork' and 'vfork' system call with syntax and example.

fork FUNCTION

An existing process can create a new one by calling the forkfunction.

```
#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 forkis 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.

The parent and the child share the text segment .

vfork FUNCTION

- ✓ The function vforkhas 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 execor exit. When the child calls either of these functions, the parent resumes.

Example of vforkfunction

```
#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. 2b) Write a program to explain setjmp and longjmp APIs .

```
// A simple C program to demonstrate working of setjmp() and longjmp()
#include<stdio.h>
#include<setjmp.h>
jmp_buf buf;
void func()
{    printf("Welcome to GeeksforGeeks\n");
    // Jump to the point setup by setjmp
    longjmp(buf, 1);
    printf("Geek2\n");
}
```


```
// A simple C program to demonstrate working of setjmp() and longjmp()
#include<stdio.h>
#include<setjmp.h>
jmp_buf buf;
void func()
{    printf("Welcome to GeeksforGeeks\n");
    // Jump to the point setup by setjmp
    longjmp(buf, 1);
    printf("Geek2\n");
}
```

```
int main()
{
    // Setup jump position using buf and return 0
    if (setjmp(buf))
        printf("Geek3\n");
    else
    {
        printf("Geek4\n");
        func();
    }
    return 0;
}
```

Output :

Geek4

Welcome to GeeksforGeeks

Geek3

Q. 3 a) What are pipes? What are its limitations? Write a program to send a data from parent to child over a pipe

PIPES

Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.

- Historically, they have been half duplex (i.e., data flows in only one direction).
- Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

A pipe is created by calling the pipefunction.

```
#include <unistd.h>

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

Returns: 0 if OK, 1 on error.

```
int
main(void
)
{
        int         n;
        int         fd[2];
        pid_t       pid;
        char        line[MAXLINE];

        if (pipe(fd) < 0) err_sys("pipe
            error");
    if ((pid = fork()) < 0) {
            err_sys("fork error");
    } else if (pid > 0) {                   /*parent */
            close(fd[0]);
            write(fd[1], "hello world\n", 12);
    } else {                                /*child */
            close(fd[1]);
            n = read(fd[0], line, MAXLINE);
            write(STDOUT_FILENO, line,
            n);
    }
    exit(0);
}
```

Q. 3b) What are Interpreter Files? Give the difference between Interpreter Files and Interpreter.

INTERPRETER FILES

These files are text files that begin with a line of the form

#! pathname [optional-argument]

The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line

#!/bin/sh

The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., PATH is not used). The recognition of these files is done within the kernel as part of processing the exec system call. The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file. Be sure to differentiate between the interpreter filea text file that begins with #!and the interpreter, which is specified by the pathname on the first line of the interpreter file.

A program that execs an interpreter file

```
#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) {                    /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) <
            0) err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0) /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Output:

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

Q. 4 a) Discuss popen and pclose functions with syntax and proper example.

popenAND pcloseFUNCTIONS

Since a common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has historically provided the popen and pclose functions. These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

Returns: file pointer if OK, NULLon error

```
int pclose(FILE *fp);
```

Returns: termination status of cmdstring, or 1 on error

The function popendoes a forkand execto execute the cmdstring, and returns a standard I/O file pointer. If type is "r", the file pointer is connected to the standard output of cmdstring

Figure 15.9. Result of fp = popen(cmdstring, "r")

If type is "w", the file pointer is connected to the standard input of cmdstring, as shown:

Figure 15.10. Result of fp = popen(cmdstring, "w")

Q. 4 b) What is FIFO? Explain the client server communication using FIFO

Example Client-Server Communication Using a FIFO

- FIFO's can be used to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
- This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server communication is how to send replies back from the server to each client.
- A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client'sprocess ID.
- For example, the server can create a FIFO with the name /vtu/ ser.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the file system.
- The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

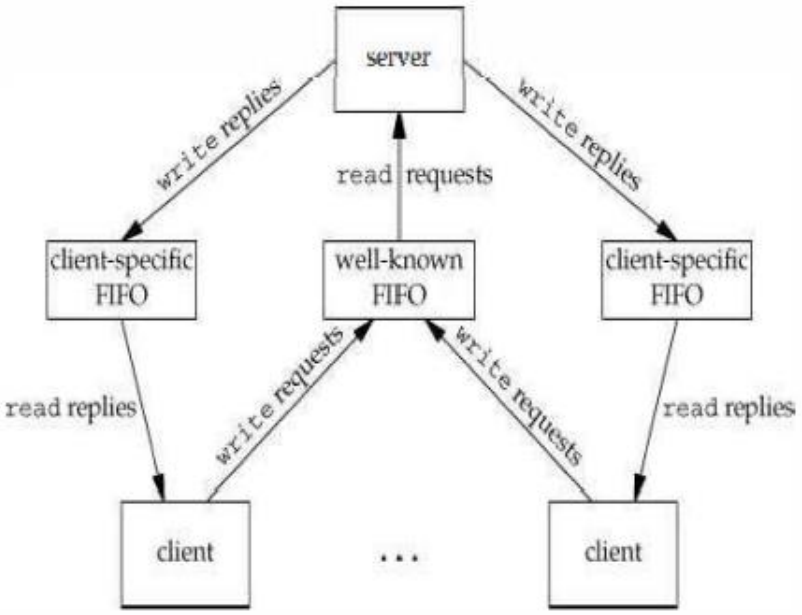


Figure 15.23. Client-server communication using FIFOs

Q. 5 a) Write a program to set up a signal hander for SIGINT and SIGALARM

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

```
cout<<"catch_sig:"<<signum<<endl;
}
/*main function*/
int main ( )
{

signal(SIGINT, catch_sig);
signal(SIGALARM,catch_sig);
pause( );

}
```

Q.5 b) What are signals? Mention different sources of signals with name and description.

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.

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

Q. 6 a) What is daemon process? Enlist its characteristics and coding rules.

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down.

DAEMON CHARACTERISTICS

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

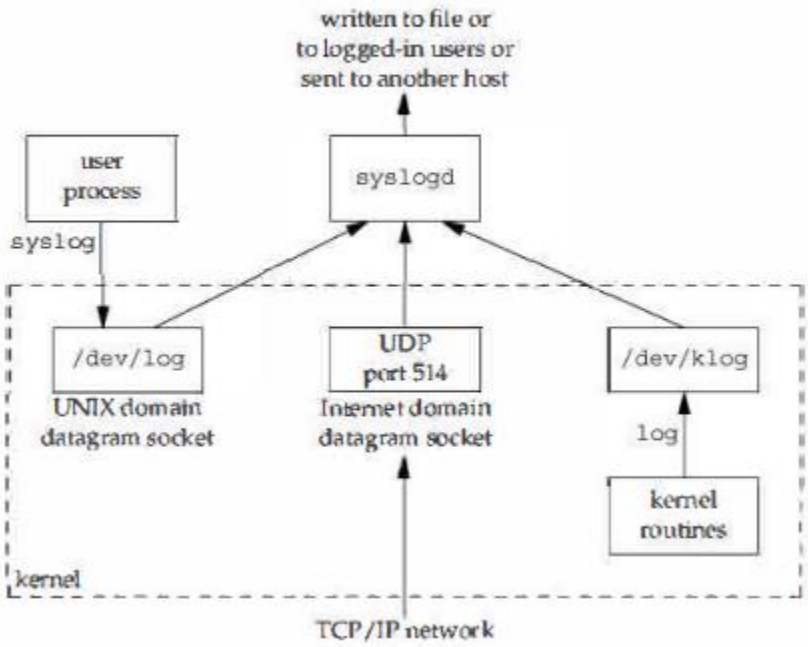
- **Call umask to set the file mode creation mask to 0**. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions.
- **Call fork and have the parent exit**. This does several things. 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.

Q.6 b) What is error logging? With a neat block schematic discuss the error login facility in BSD

ERROR LOGGING

One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, since on many workstations, the console device runs a windowing system. A central daemon error-logging facility is required.

Figure 13.2. The BSD `syslog` facility



There are three ways to generate log messages:

- Kernel routines can call the logfunction. These messages can be read by any user process that opens and
  reads the /dev/klogdevice.
- Most user processes (daemons) call the syslog(3) function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
- A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the syslog function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.