USN | | | | | | | | | |

CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 5 – February 2022

Sub:	Unix Programming				Sub Code:	18CS56	Branch:	ISE
Date:	7/2/2022	Duration:	90 min's	Max Marks: 50	Sem/Sec:	V A, B & C		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1	Describe the kill( ) and alarm( ) API with example.	10	CO4	L2
2	Write a short notes on Message Queue & Semaphores.	10	CO4	L1
3	Discuss wait and waitpid APIs with their prototype. Mention the differences between wait and waitpid.	10	CO3	L2
4	Describe with a neat diagram, how a process can be initiated and how it can be terminated.	10	CO3	L1
5	Describe the Unix Kernel support for the process considering parent and child process. Show the related data structures.	10	CO3	L2
6	Mention the syntax of getrlimit and setrlimit functions. Apply the same for some real time example.	10	CO4	L3

Faculty Signature　　　　CCI Signature　　　　HOD Signature

Scheme of Evaluation

Internal Assessment Test 5 – Feb 22

Sub:	UNIX Programming							**Code:**	18CS56
Date:	7/2/2022	Duration:	90mins	Max Marks:	50	**Sem:**	V	**Branch:**	ISE

Note: **Answer Any five full questions.**

Question #		Description	Marks Distribution		Max Marks
1		Describe the kill( ) and alarm( ) API with example.	2 * 5M	10M	10M
2		Write a short notes on Message Queue & Semaphores.	2*5M	10M	10M
3		Discuss wait and waitpid APIs with their prototype. Mention the differences between wait and waitpid.	2* 4M 2M	10M	10M
4		Describe with a neat diagram, how a process can be initiated and how it can be terminated.	3M + 7M	10M	10M
5		Describe the Unix Kernel support for the process considering parent and child process. Show the related data structures.	3M+ 7M	10M	10M
6		Mention the syntax of getrlimit and setrlimit functions. Apply the same for some real time example.	3.5+3.5+3	10M	10M

Scheme Of Evaluation Internal Assessment Test 5 – FEB 2022

Sub:	UNIX Programming							**Code:**	18CS56
Date:	7/2/2022	Duration:	90mins	Max Marks:	50	**Sem:**	V	**Branch:**	ISE

Note: Answer Any full five questions

1. Describe the kill( ) and alarm( ) API with example. 10M

KILL

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 signal_num argument is the integer value of a signal to be sent to one or more processes designated by pid. The possible values of pid and its use by the kill API are:

pid > 0	The signal is sent to the process whose process ID is pid.
pid == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal.
pid == 1	The signal is sent to all processes on the system for which the sender has permission to send the signal.

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

```
#include<io
stream.h>
#include<st
dio.h>
#include<un
istd.h>
#include<st
ring.h>
#include<si
gnal.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

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,&actio
        n,0)==-1)
        {
              perror("sig
              action");
              return -1;
        }
(void) alarm (timer);
        (void)
        pause( );
        return 0;
   }
```

2. Write a short notes on Message Queue & Semaphores.

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a queue and its identifier a queue ID.

A new queue is created or an existing queue opened by `msgget`. New messages are added to the end of a queue by `msgsnd`. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to `msgsnd` when the message is added to a queue. Messages are fetched from a queue by `msgrcv`. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following `msqid_ds` structure associated with it:

```
struct msqid_ds
{
        struct ipc_perm  msg_perm;     /* see Section 15.6.2 */
        msgqnum_t        msg_qnum;     /* # of messages on queue */
        msglen_t         msg_qbytes;   /* max # of bytes on queue */
        pid_t            msg_lspid;    /* pid of last msgsnd() */
        pid_t            msg_lrpid;    / pid of last msgrcv() */

        time_t           msg_stime;   /*   last-msgsnd()
        time */ time_t   msg_rtime;   /*   last-msgrcv()
        time */ time_t   msg_ctime; /* last-change time
        */
        .
        .
        .
};
```

This structure defines the current status of the queue.

The first function normally called is `msgget` to either open an existing queue or create a new queue.

```
#include <sys/msg.h>
int msgget(key_t key, int flag);
```

Returns: message queue ID if OK, 1 on error

When a new queue is created, the following members of the `msqid_ds` structure are initialized.

- ✓ The `ipc_perm` structure is initialized. The `mode` member of this structure is set to the corresponding permission bits of flag.
- ✓ `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are all set to 0.
- ✓ `msg_ctime` is set to the current time.
- ✓ `msg_qbytes` is set to the system limit.

On success, `msgget` returns the non-negative queue ID. This value is then used with the other three message queue functions.

The `msgctl` function performs various operations on a queue.

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf );
```

Returns: 0 if OK, 1 on error.

The cmd argument specifies the command to be performed on the queue specified by msqid.

Table 9.7.2 POSIX:XSI values for the cmd parameter of msgctl.	
cmd	**description**
IPC_RMID	remove the message queue msqid and destroy the corresponding msqid_ds
IPC_SET	set members of the msqid_ds data structure from buf
IPC_STAT	copy members of the msqid_ds data structure into buf

Data is placed onto a message queue by calling `msgsnd`.

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Returns: 0 if OK, 1 on error.

Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if nbytes is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg{
  long  mtype;      /* positive message type */
  char  mtext[512]; /* message data, of length nbytes */
};
```

SEMAPHORES

A semaphore is a counter used to provide access to a shared data object for multiple processes.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

A common form of semaphore is called a ***binary semaphore***. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The undo feature that we describe later is supposed to handle this.

The kernel maintains a `semid_ds` structure for each semaphore set:

```
struct semid_ds {
    struct ipc_perm  sem_perm;  /* see Section
    15.6.2 */ unsigned short    sem_nsems; /* # of
    semaphores in set */ time_t sem_otime; /* last-
    semop() time */
    time_t           sem_ctime; /* last-change time */
    .
    .
    .
  };
```

Each semaphore is represented by an anonymous structure containing at least the following members:

```
struct {

    unsigned short   semval;   /* semaphore value,
    always >= 0 */ pid_t       sempid;      /* pid for
    last operation */
    unsigned short   semncnt;  /* # processes awaiting
    semval>curval */ unsigned short semzcnt;     /* #
    processes awaiting semval==0 */
    .
    .
    .
  };
```

The first function to call is semget to obtain a semaphore ID.

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
```

Returns: semaphore ID if OK, 1 on error

When a new set is created, the following members of the semid_ds structure are initialized.

- The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding permission bits of flag.
- sem_otime is set to 0.
- sem_ctime is set to the current time.
- sem_nsems is set to nsems.

The number of semaphores in the set is nsems. If a new set is being created (typically in the server), we must specify nsems. If we are referencing an existing set (a client), we can specify nsems as 0.

The semctl function is the catchall for various semaphore operations.

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int  cmd,... /* union semun arg */);
```

The fourth argument is optional, depending on the command requested, and if present, is of type `semun`, a union of
various command-specific arguments:

```
union semun
{
  int   val;              /* for SETVAL */
  struct semid_ds *buf;    /* for IPC_STAT and
  IPC_SET */ unsigned short*array;/* for GETALL
  and SETALL */
};
```

3. Discuss wait and waitpid APIs with their prototype. Mention the differences between wait and waitpid.

wait AND waitpid FUNCTIONS

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event - it can happen at any time while the parent is running - this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.

A process that calls wait or waitpid can:

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

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

wait(int *statloc);

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

Both return: process ID if OK, 0 (see later), or 1 on error.

The differences between these two functions are as follows.
The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.
If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates.

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

Program to Demonstrate various exit statuses

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

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

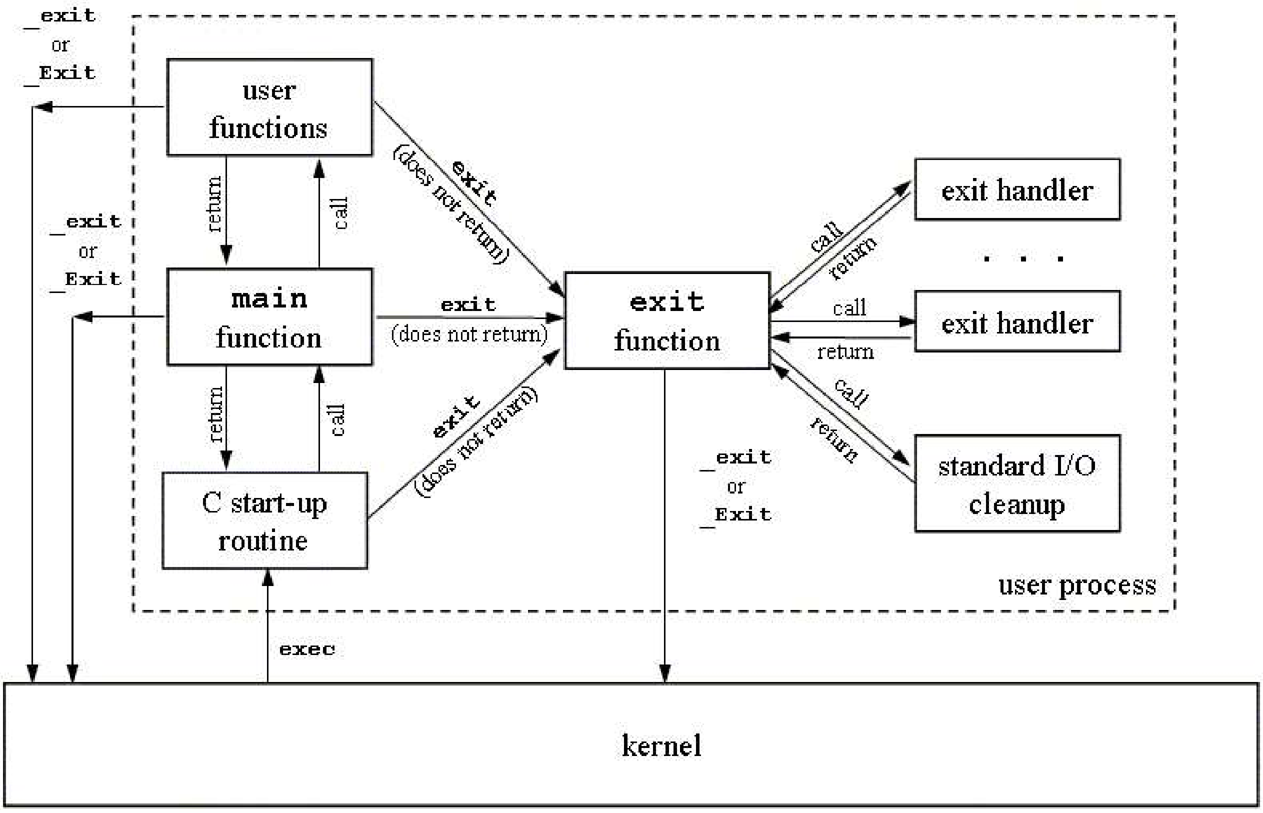
- The waitpid function lets us wait for one particular process, whereas the wait function returns the status of any terminated child. We'll return to this feature when we discuss the popen function.
- The waitpid function provides a nonblocking version of wait. There are times when we want to fetch a child's status, but we don't want to block.
- The waitpid function provides support for job control with the WUNTRACED and WCONTINUED options.

4. Describe with a neat diagram, how a process can be initiated and how it can be terminated.

A C program starts execution with a function called main. The prototype for the main function is:
int main(int argc, char *argv[]);
where argc is the number of command-line arguments, and argv is an array of pointers to the arguments.

When a C program is executed by the kernel by one of the exec functions, a special start-up routine is called before the main function is called. The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler. This start-up routine takes values from the kernel, the command-line arguments and the environment and sets things up so that the main function is called.

The following figure summarizes how a C program is started and the various ways it can terminate.



Process Termination:

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

- 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.

Abnormal termination occurs in three ways:

- Calling abort.
- Receipt of a signal.
- Response of the last thread to a cancellation request.

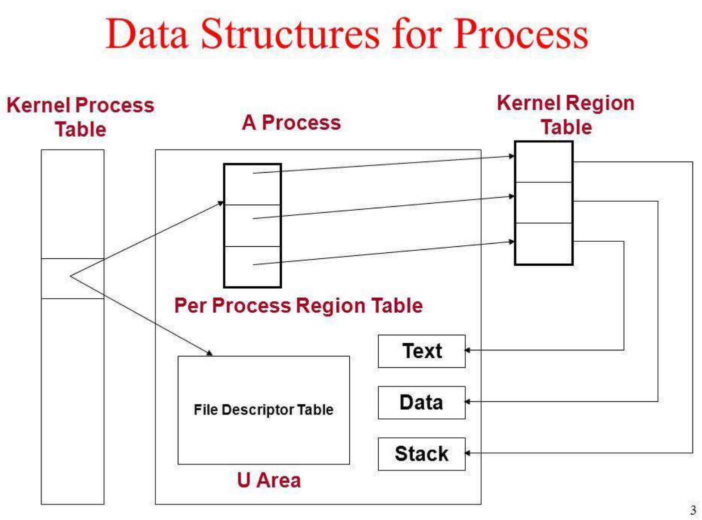
5. Describe the Unix Kernel support for the process considering parent and child process. Show the related data structures.

The data structure and execution of processes are dependent on operating system implementation.
A UNIX process consists minimally of a text segment, a data segment and a stack segment. A segment is an area of memory that is managed by the system as a unit.

- A text segment consists of the program text in machine executable instruction code format.

- The data segment contains static and global variables and their corresponding data.
- A stack segment contains runtime variables and the return addresses of all active functions for a process.

UNIX kernel has a process table that keeps track of all active process present in the system. Some of these processes belongs to the kernel and are called as “system process”. Every entry in the process table contains pointers to the text, data and the stack segments and also to U-area of a process. U-area of a process is an extension of the process table entry and contains other process specific data such as the file descriptor table, current root and working directory inode numbers and set of system imposed process limits.



All processes in UNIX system expect the process that is created by the system boot code, are created by the fork system call. After the fork system call, once the child process is created, both the parent and child processes resumes execution. When a process is created by fork, it contains duplicated copies of the text, data and stack segments of its parent as shown in the Figure below. Also it has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

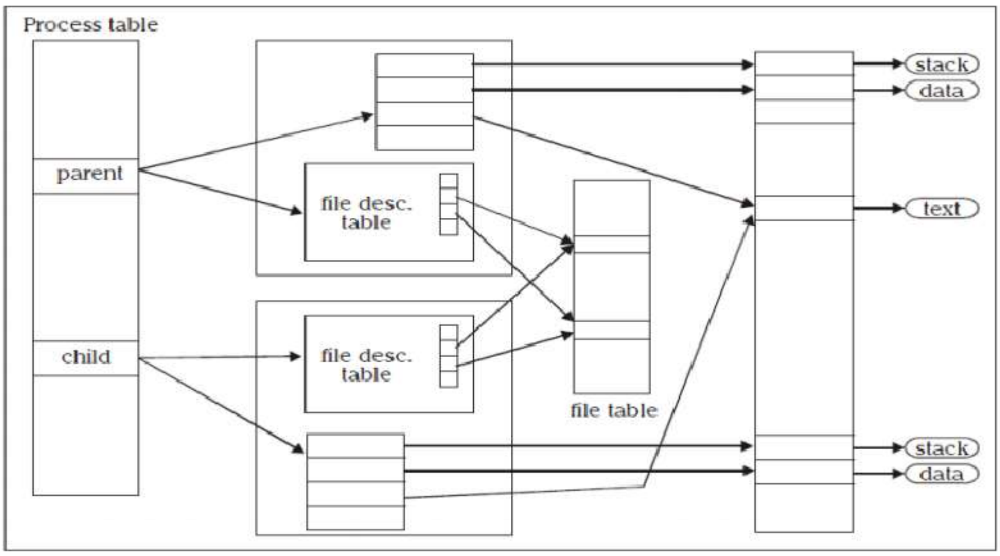


Fig: Parent & Child relationship after fork

6. Mention the syntax of getrlimit and setrlimit functions. Apply the same for some real time example.

getrlimit AND setrlimit FUNCTIONS

Every process has a set of resource limits, some of which can be queried and changed by the geTRlimit and setrlimit functions.

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

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

Both return: 0 if OK, nonzero on error

Each call to these two functions specifies a single resource and a pointer to the following structure:

```
struct rlimit
{
rlim_t rlim_cur; /* soft limit: current limit */
rlim_t rlim_max; /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

- A process can change its soft limit to a value less than or equal to its hard limit.
- A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
- Only a superuser process can raise a hard limit.

Example: Print the current resource limits

```
#include "apue.h"
#if defined(BSD) || defined(MACOS)
#include <sys/time.h>
#define FMT "%10lld "
#else
#define FMT "%10ld "
#endif
#include <sys/resource.h>
#define doit(name) pr_limits(#name, name)
static void pr_limits(char *, int);
int main(void)
{
#ifdef RLIMIT_AS
doit(RLIMIT_AS);
#endif
doit(RLIMIT_CORE);
doit(RLIMIT_CPU);
doit(RLIMIT_DATA);
doit(RLIMIT_FSIZE);
#ifdef RLIMIT_LOCKS
doit(RLIMIT_LOCKS);
#endif
#ifdef RLIMIT_MEMLOCK
doit(RLIMIT_MEMLOCK);
#endif
doit(RLIMIT_NOFILE);
#ifdef RLIMIT_NPROC
doit(RLIMIT_NPROC);
#endif
#ifdef RLIMIT_RSS
```

```
doit(RLIMIT_RSS);
#endif
#ifdef RLIMIT_SBSIZE
doit(RLIMIT_SBSIZE);
#endif
doit(RLIMIT_STACK);
#ifdef RLIMIT_VMEM
doit(RLIMIT_VMEM);
#endif
exit(0);
}
static void pr_limits(char *name, int resource)
{
struct rlimit limit;
if (getrlimit(resource, &limit) < 0)
err_sys("getrlimit error for %s", name);
printf("%-14s ", name);
if (limit.rlim_cur == RLIM_INFINITY)
printf("(infinite) ");
else
printf(FMT, limit.rlim_cur);
if (limit.rlim_max == RLIM_INFINITY)
printf("(infinite)");
else
printf(FMT, limit.rlim_max);
putchar((int)'\n');
}
```

****