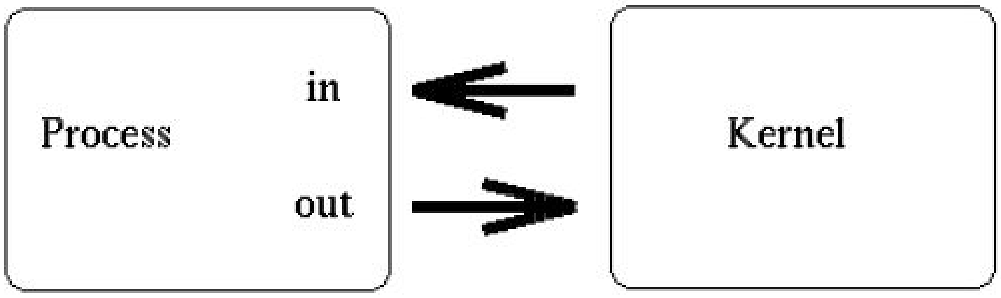
Hands-on Exercise 2

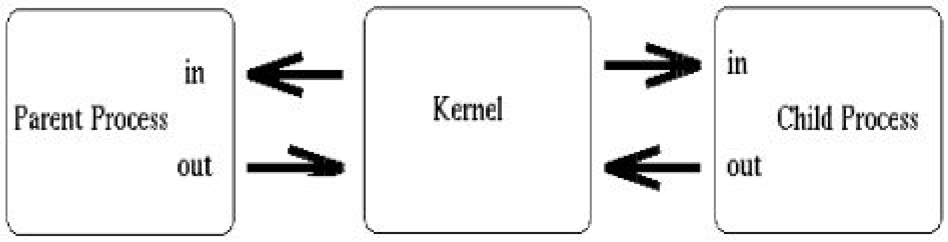
4st October 2020

PIPE

A pipe is a method of connecting the standard output of one process to the standard input of another. Example ls -l | grep file



When a process creates a pipe, the kernel sets up two file descriptors for use by the pipe. One descriptor is used to allow a path of input into the pipe (write), while the other is used to obtain data from the pipe (read). At this point, the pipe is of little practical use, as the creating process can only use the pipe to communicate with itself.



The creating process typically forks a child process. Since a child process will inherit any open file descriptors from the parent, we now have the basis for multi process communication (between parent and child).

Creating a pipe

Check for **man 2 pipe**. fd[0] is set up for reading, fd[1] is set up for writing.

SYSTEM CALL: pipe();

PROTOTYPE: int pipe(int fd[2]);

RETURNS: 0 on success

-1 on error: errno = EMFILE (no free descriptors)

EMFILE (system file table is full)

EFAULT (fd array is not valid)

14. Write a simple program to create a pipe, write to the pipe, read from pipe and display on the monitor.

```
int pipefd[2];
pipe(pipefd)
write(pipefd[1], msg, sizeof(msg));
read(pipefd[0], buff, sizeof(msg));
```

15. Write a simple program to send some data from the parent to the child process.

```
int pipefd[2];
pipe(pipefd)
int child_pid = fork();
if(child_pid > 0){
    // parent process write
}else{
   // child process read
}
```

16. Write a program to send and receive data from parent to child vice versa. Use two way communication.

```
int pipe1[2], pipe2[2];
pipe(pipe1);
pipe(pipe2);
int child_pid = fork();
if(child_pid > 0){
    // close read1 and close write2, parent process write1
    // read from read2
}else{
   // close read2 and write1, child process read1
   // write from write2
}
```

17. Write a program to execute ls -l | wc. use dup/dup2/fcntl

We know for dup, when we duplicate a file descriptor it is duplicated with the lowest fd.

```
int pipe1[2];
pipe(pipe1);
int child_pid = fork();
if(child_pid == 0){
    // close STDOUT, dup pipe[1], close rest and execl command
}else{
    // close STDIN, dup pipe[0], close rest and execl command
}
```

18. Write a program to find out the total number of directories on the pwd. Execute ls -l | grep ^d | wc ? Use only dup2.

```
int pipe1[2], pipe2[2];
pipe(pipe1);
pipe(pipe2);
int child1 = fork();
if(child1 == 0){
    // close STDOUT, dup2(pipe1[1],1), close rest and execl
command
}else{
    if(!fork()){
        // close STDIN, dup2(pipe1[0],0), close STDOUT,
dup2(pipe2[1],1), close rest and execl command
    }
    else{
        // close STDIN, dup2(pipe2[0],0) close rest and execl
command
    }
}
```

19. Create a FIFO file by, a. mknod command, b. mkfifo command, c. use strace command to find out, which command (mknod or mkfifo) is better, d. mknod system call, e. mkfifo library function

```
strace -c mkfifo <filename>
strace -c mknod <filename> p
```

20. Write two programs so that both can communicate by FIFO -Use one way communication.

Create two processes, where in 20a you create a pipe using mknod or mkfifo and then open that file in write only mode while writing some data to it. While in 20b you open the same pipe in read only mode and read that written stuff here.

21. Write two programs so that both can communicate by FIFO -Use two way communications.

Create two processes, where in 21a you create 2 pipes using mknod or mkfifo and then open those files one in write only mode and another in read only mode. Writing some data to pipe1 and reading some data from pipe2. While in 21b you open those same pipes in read only mode and write only mode respectively. And read from pipe1 and write in pipe2.

```
mkfifo("21a_fifoFile",
S_IWGRP|S_IRGRP|S_IRUSR|S_IWUSR|S_IROTH);
mkfifo("21b_fifoFile",
S_IWGRP|S_IRGRP|S_IRUSR|S_IWUSR|S_IROTH);
fdw = open("21a_fifoFile", O_WRONLY);
fdr = open("21b_fifoFile", O_RDONLY);
write(fdw, buff_w, sizeof(buff_w));
read(fdr, buff_r, sizeof(buff_r));
```

22. Write a program to wait for data to be written into FIFO within 10 seconds, use select system call with FIFO.

Normally, blocking occurs on a FIFO. In other words, if the FIFO is opened for reading, the process will "block" until some other process opens it for writing. This action works vice-versa as well. If this behavior is undesirable, the O_NONBLOCK flag can be used in an open() call to disable the default blocking action.

ndfs - By specifying the highest descriptor we're interested in, the kernel can avoid going through hundreds of unused bits in the three descriptor sets, looking for bits that are turned on.

Create a pipe and then parameters of the select system call and use it to check if the data is available, if yes print it on the screen. For this to achieve, create two files in 22a.c you keep check of the data in the pipe, and from 22b.c you write the data in the pipe.

```
fd = open("22_fifoFile", O_NONBLOCK|O_RDONLY);
FD_ZERO(&rfds);
FD_SET(fd, &rfds);
if(select(4, &rfds, NULL, NULL, &timeSet)){
   // read the data in fd, and then print the received data
}

21b
fd = open("22_fifoFile", O_WRONLY);
write(fd, "Hey there", sizeof("Hey there"));
```

23. Write a program to print the maximum number of files that can be opened within a process and size of a pipe (circular buffer).

To get these configuration information, use sysconf, man 3 sysconf

To get the file configuration values, use pathconf, man 3 pathconf

```
sysconf(_SC_OPEN_MAX);
pathconf("pipename", _PC_PIPE_BUF);
```

MESSAGE QUEUE

Message queues can be best described as an internal linked list within the kernel's addressing space. Messages can be sent to the queue in order and retrieved from the queue in several different ways.

man 7 sysvipc

ftok - convert a pathname and a project identifier to a System V IPC key

The ftok() function uses the identity of the file named by the given pathname (which must refer to an existing, accessible file) and the least significant 8 bits of proj_id (which must be nonzero) to generate a key_t type System V IPC key, suitable for use with msgget(2), semget(2), or shmget(2).

man 3 ftok

ipcs -q

24. Write a program to create a message queue and print the key and message queue id.

To access any IPC we have to first create a key, which can be done using the ftok library function. msgget system call will be used to create a message queue and print the queue id.

The msgget() system call returns the System V message queue identifier associated with the value of the key argument. It may be used either to obtain the identifier of a previously created message queue or to create a new set. IPC_CREAT is used to create a new message queue.

man 2 msgget

```
key = ftok(".", 67);
msgid = msgget(key, IPC_CREAT|IPC_EXCL|0744);
```

25. Write a program to print a message queues (use msqid_ds and ipc_perm structures)

a. access permission

b. uid, gid

c. time of last message sent and received

d. time of last change in the message queue

d. size of the queue

f. number of messages in the queue

g. maximum number of bytes allowed

h. pid of the msgsnd and msgrcv

int msgctl(int msqid, int cmd, struct msqid_ds *buf);

msgctl() performs the control operation specified by cmd on the System V message queue with identifier msqid.

man 2 msgctl

IPC_STAT - Copy information from the kernel data structure associated with msqid into the msqid_ds structure pointed to by buf. The caller must have read permission on the message queue.

```
struct msqid_ds mq;
key = ftok(".", 67);
msgid = msgget(key, 0);
msgctl(msgid, IPC_STAT, &mq);
```

26. Write a program to send messages to the message queue. Check $ipcs -q

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

The msgsnd() system call is used to send messages to a System V message queue. The calling process must have write permission on the message queue in order to send a message.

The first argument to msgsnd is our queue identifier, returned by a previous call to msgget. The second argument, msgp, is a pointer to our redeclared and loaded message buffer. The msgsz argument contains the size of the message in bytes, excluding the length of the message type (4 byte long).

man 2 msgsnd

IPC_NOWAIT - If the message queue is full, then the message is not written to the queue, and control is returned to the calling process. If not specified, then the calling process will suspend (block) until the message can be written.

```
struct msgbuf {
    long mtype;       /* message type, must be > 0 */
    char mtext[1024];    /* message data */
}msg;
// create a key
// get the details of the message queue based on key
// set the msg type and message
// use msgid received from above step to set parameters in
msgsnd
msgsnd(msgid, &msg, strlen(msg.mtext)+1, 0);
```

The ability to assign a given message a *type*, essentially gives you the capability to *multiplex* messages on a single queue. For instance, client processes could be assigned a magic number, which could be used as the message type for messages sent from a server process. The server itself could use some other number, which clients could use to send messages to it. In another scenario, an application could mark error messages as having a message type of 1, request messages could be type 2, etc. The possibilities are endless.

27. Write a program to receive messages from the message queue.

a. with 0 as a flag

b. with IPC_NOWAIT as a flag

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

The msgrcv() system call is used to receive messages from a System V message queue. The calling process must have read permission to receive a message.

The first argument is used to specify the queue to be used during the message retrieval process (should have been returned by an earlier call to msgget).

The second argument (msgp) represents the address of a message buffer variable to store the retrieved message at. The third argument (msgsz) represents the size of the message buffer structure, excluding the length of the mtype member.

The fourth argument (mtype) specifies the *type* of message to retrieve from the queue. The kernel will search the queue for the oldest message having a matching type, and will return a copy of it in the address pointed to by the msgp argument. One special case exists. If the mtype argument is passed with a value of zero, then the oldest message on the queue is returned, regardless of type.

man 2 msgrcv

If the **msgflag is 0** and the requested message type is not in the message queue, the process will wait till some sender sends the message.

If **IPC_NOWAIT** is passed as a flag, and no messages are available, the call returns ENOMSG to the calling process.

```
//define the structure for msgbuf
// create a key
// get the details of the message queue based on key
// choose the message type
// use msgid received from above step to set parameters in
msgrcv
msgrcv(msgid,&msg,sizeof(msg.mtext),msg.mtype,IPC_NOWAIT);
```

28. Write a program to change the existing message queue permission. (use msqid_ds structure)

IPC_SET - Write the values of some members of the msqid_ds structure pointed to by buf to the kernel data structure associated with this message queue, updating also its msg_ctime member. The following members of the structure are updated: msg_qbytes, msg_perm.uid, msg_perm.gid, and (the least significant 9 bits of) msg_perm.mode.

Use msgctl to set the permissions in struct ipc_perm i.e. mode and which indeed can be changed by struct msqid_ds

man 2 msgctl

```
// create a key
// get the id of the message queue based on key
// get the stat for the message id
// make changes in the msgid_ds mode
// set the changes you made in mode using msgctl
mq.msg_perm.mode = 0777
msgctl(msgid, IPC_SET, &mq);
// verify the same
```

29. Write a program to remove the message queue.

IPC_RMID - Immediately remove the message queue, awakening all waiting reader and writer processes (with an error return and errno set to EIDRM).

In msgctl() the calling process must have appropriate privileges or its effective user ID must be either that of the creator or owner of the message queue. The third argument to msgctl() is ignored in this case.

man 2 msgctl

```
// create a key
// get the id of the message queue based on key
msgctl(msgid, IPC_RMID, 0);
```

SHARED MEMORY

Shared memory can best be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process. This is by far the fastest form of IPC, because there is no intermediation (i.e. a pipe, a message queue, etc). Instead, information is mapped directly from a memory segment, and into the addressing space of the calling process. A segment can be created by one process, and subsequently written to and read from by any number of processes.

man 7 sysvipc

ipcs -m

30. Write a program to create a shared memory.

a. write some data to the shared memory

b. attach with O_RDONLY and check whether you are able to overwrite.

c. detach the shared memory

d. remove the shared memory

The procedure is the same for shared memory as we had in message queue, first we create a unique key which distinguishes the shared memory, The unique key can be created using ftok.

int shmget(key_t key, size_t size, int shmflg);

In order to create a new message queue, or access an existing queue, the shmget() system call is used. The first argument to shmget() is the key value (in our case returned by a call to ftok()). This key value is then compared to existing key values that exist within the kernel for other shared memory segments. At that point, the open or access operation is dependent upon the contents of the shmflg argument.

If IPC_CREAT is used alone, shmget() either returns the segment identifier for a newly created segment, or returns the identifier for a segment which exists with the same key value. If IPC_EXCL is used along with IPC_CREAT, then either a new segment is created, or if the segment exists, the call fails with -1. IPC_EXCL is useless by itself, but when combined with IPC_CREAT, it can be used as a facility to guarantee that no existing segment is opened for access.

man 2 shmget

void *shmat(int shmid, const void *shmaddr, int shmflg);

shmat() attaches the System V shared memory segment identified by shmid to the address space of the calling process.

If the addr argument is zero (0), the kernel tries to find an unmapped region. This is the recommended method. An address can be specified, but is typically only used to facilitate proprietary hardware or to resolve conflicts with other apps. The SHM_RND flag can be OR'd into the flag argument to force a passed address to be page aligned (rounds down to the nearest page size).

In addition, if the SHM_RDONLY flag is OR'd in with the flag argument, then the shared memory segment will be mapped in, but marked as readonly. The return value of shmat is void *pointer. The pointer to void is actually part of the ANSI-C standard; and it can be assigned the value of any pointer type.

man 2 shmat

int shmdt(const void *shmaddr);

shmdt() detaches the shared memory segment located at the address specified by shmaddr from the address space of the calling process. The to-be-detached segment must be currently attached with shmaddr equal to the value returned by the attaching shmat() call.

man 2 shmdt

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

shmctl() performs the control operation specified by cmd on the System V shared memory segment whose identifier is given in shmid.

IPC_RMID - Marks a segment for removal. The IPC_RMID command doesn't actually remove a segment from the kernel. Rather, it marks the segment for removal. The actual removal itself occurs when the last process currently attached to the segment has properly detached it. Of course, if no processes are currently attached to the segment, the removal seems immediate.

To properly detach a shared memory segment, a process calls the *shmdt* system call.

man 2 shmctl

```
// create a key
key_t key = ftok(“.”, 67);
// get the id of the shared memory based on key
shmid = shmget(key, 1024, IPC_CREAT|0744);
// attach the shmid to address space of calling process
ptr = shmat(shmid, (void *)0, SHM_RDONLY);
// write some data on this ptr
// set the shmflag to SHM_RDONLY in shmat
scanf(“%[^\n]”, ptr);
// try writing some data
// detach the ptr from shared memory
shmdt(ptr)
//remove the shared memory
shmctl(shmid, IPC_RMID, NULL);
```

SEMAPHORES

Semaphores can best be described as counters used to control access to shared resources by multiple processes. They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it.

Assume that in our corporate print room, we have 5 printers online. Our print spool manager allocates a semaphore set with 5 semaphores in it, one for each printer on the system. Since each printer is only physically capable of printing one job at a time, each of our five semaphores in our set will be initialized to a value of 1 (one), meaning that they are all online, and accepting requests.

John sends a print request to the spooler. The print manager looks at the semaphore set, and finds the first semaphore which has a value of one. Before sending John's request to the physical device, the print manager *decrements* the semaphore for the corresponding printer by a value of negative one (-1). Now, that semaphore's value is zero. In the world of System V semaphores, a value of zero represents 100% resource utilization on that semaphore. In our example, no other request can be sent to that printer until it is no longer equal to zero.

When John's print job has completed, the print manager *increments* the value of the semaphore which corresponds to the printer. Its value is now back up to one (1), which means it is available again. Naturally, if all 5 semaphores had a value of zero, that would indicate that they are all busy printing requests, and that no printers are available.

man 7 sysvipc

ipcs -s

31. Write a program to create a semaphore and initialize value to the semaphore.

a. create a binary semaphore

b. create a counting semaphore

Procedure is the same where we first create a unique key which distinguishes our semaphore, The unique key can be created using ftok.

int semget(key_t key, int nsems, int semflg);

The semget() system call returns the System V semaphore set identifier associated with the argument key.

The first argument to semget() is the key value (in our case returned by a call to ftok()). This key value is then compared to existing key values that exist within the kernel for other semaphore sets. At that point, the open or access operation is dependent upon the contents of the semflg argument. nsems specifies the number of semaphores set that will be created.

IPC_CREAT - Create the semaphore set if it doesn't already exist in the kernel.

IPC_EXCL - When used with IPC_CREAT, fail if the semaphore set already exists.

man 2 semget

int semctl (int semid, int semnum, int cmd, union semun arg); man 2 semctl

The *semctl* system call is used to perform control operations on a semaphore set. This call is analogous to the *msgctl* system call which is used for operations on message queues. If you compare the argument lists of the two system calls, you will notice that the list for *semctl* varies slightly from that of *msgctl*. semnum is the index of the semaphore set we are working on.

The cmd argument represents the command to be performed against the set.

IPC_STAT - Retrieves the semid_ds structure for a set, and stores it in the address of the buf argument in the semun union.

IPC_SET - Sets the value of the ipc_perm member of the semid_ds structure for a set. Takes the values from the buf argument of the semun union.

IPC_RMID - Removes the set from the kernel.

SETVAL - Sets the value of an individual semaphore within the set to the *val* member of the union.

```
/* arg for semctl system calls. */
union semun {
     int val;                 /* value for SETVAL */
     struct semid_ds *buf;    /* buffer for IPC_STAT & IPC_SET
*/
     ushort *array;           /* array for GETALL & SETALL */
};
```

val - Used when the SETVAL command is performed. Specifies the value to set the semaphore to.

buf - Used in the IPC_STAT/IPC_SET commands. Represents a copy of the internal semaphore data structure used in the kernel.

array - A pointer used in the GETALL/SETALL commands. Should point to an array of integer values to be used in setting or retrieving all semaphore values in a set.

```
union semun arg;
// set the key
Key_t key = ftok('.', 112);
semid = semget(key, 1, IPC_CREAT|0644);
// set the val in arg to the semaphore value
arg.val = 1; // more than 1 for counting semaphore
semctl(semid, 0, SETVAL, arg);
```

32. Write a program to implement semaphore to protect any critical section.

a. rewrite the ticket number creation program using semaphore

b. protect shared memory from concurrent write access

c. protect multiple pseudo resources (may be two) using counting semaphore

d. remove the created semaphore

int semop (int semid, struct sembuf *sops, unsigned nsops);

The first argument to semget() is the key value (in our case returned by a call to semget). The second argument (sops) is a pointer to an array of *operations* to be performed on the semaphore set, while the third argument (nsops) is the number of operations in that array.

```
/* semop system call takes an array of these */
struct sembuf {
       ushort  sem_num;        /* semaphore index in array */
       short   sem_op;         /* semaphore operation */
       short   sem_flg;        /* operation flags */
};
```

sem_num - The number of the semaphore you wish to deal with

sem_op - The operation to perform (positive, negative, or zero)

sem_flg - Operational flags

If **IPC_NOWAIT** is not specified, then the calling process sleeps until the requested amount of resources are available in the semaphore (another process has released some).

```
union semun arg;
// set the key
// get the semid based on the key
// set the val in arg to the semaphore value
arg.val = 1; // more than 1 for counting semaphore
semctl(semid, 0, SETVAL, arg);
struct sembuf sops = {0, -1, 0};
semop(semid, &sops, 1)
// Inside the critical section
sops.sem_op = 1;
semop(semid, &sops, 1)
// Lock released
semctl(semid, 0, IPC_RMID, arg) //delete the semaphore
```

SOCKET PROGRAMMING

The following tasks are done at client side:

1. · Create a socket for communication
2. · Configure TCP protocol with IP address of server and port number
3. · Connect with server through socket
4. · Wait for acknowledgement from server
5. · Send message to the server
6. · Receive message from server

The following tasks are done at server side:

1. · Create a socket for communication
2. · Bind the local port and connection address
3. · Configure TCP protocol with port number
4. · Listen for client connection
5. · Accept connection from client
6. · Send Acknowledgement
7. · Receive message from client
8. · Send message to the client

Important steps:

1. create() → Create TCP socket.
2. bind() → Bind the socket to server address.
3. listen() → put the server socket in a passive mode, where it waits for the client to approach the server to make a connection
4. accept() → At this point, connection is established between client and server, and they are ready to transfer data.

The following steps can be followed to write client side program:

- The call to the function ‘**socket()**’ creates an UNnamed socket inside the kernel and returns an integer known as socket descriptor.
- This function takes domain/family as its first argument. For the Internet family of IPv4 addresses we use **AF_INET**.
- The second argument **‘SOCK_STREAM’** specifies that the transport layer protocol that we want should be reliable i.e., it should have acknowledgement techniques. For example : TCP
- The third argument is generally left **zero** to let the kernel decide the default protocol to use for this connection. For connection oriented reliable connections, the default protocol used is TCP.
- The call to the function ‘**bind()**’ assigns the details specified in the structure **‘serv_addr’** to the socket created in the step above. The details include, the family/domain, the interface to listen on(in case the system has multiple interfaces to the network) and the port on which the server will wait for the client requests to come.
- The call to the function ‘**listen()**’ with a second argument as ’10’ specifies the maximum number of client connections that server will queue for this listening socket.
- After the call to listen(), this socket becomes a fully functional listening socket.
- In the call to **accept()**, the server is put to sleep and when for an incoming client request, the three way TCP handshake* is complete, the function accept () wakes up and returns the socket descriptor representing the client socket.
- The call to accept() is run in an infinite loop so that the server is always running and the delay or sleep of 1 sec ensures that this server does not eat up all of your CPU processing.

- As soon as the server gets a request from the client, it prepares the date and time and writes on the client socket through the descriptor returned by accept().

The following steps can be followed to write client side program:

- Even here, a socket is created through a call to **socket()** function.
- Information like the IP address of the remote host and its port is bundled up in a structure and a call to function **connect()** is made which tries to connect this socket with the socket (IP address and port) of the remote host.
- Note that here we don't bind our client socket on a particular port as client generally use port assigned by kernel as client can have its socket associated with any port but In case of server it has to be a well known socket, so known servers bind to a specific port like HTTP server runs on port 80 etc while there is no such restrictions on clients.
- Once the sockets are connected, the server sends the data (date+time) on clients socket through clients socket descriptor and client can read it through normal read call on its socket descriptor.

33. Write a program to communicate between two machines using socket.

man 7 ip

man 2 socket

man 2 bind

man 2 listen

man 2 accept

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t    s_addr;     /* address in network byte order */
};
```

man 2 socket

man 7 ip

man 2 connect

1. Write a separate program (for each time domain) to set a interval timer in 10sec and 10 microseconds

a. ITIMER_REAL

b. ITIMER_VIRTUAL

c. ITIMER_PROF

man 2 setitimer

```
struct itimerval {
  struct timeval it_interval; /* Interval for periodic timer */
  struct timeval it_value;    /* Time until next expiration */
};
struct timeval {
  time_t      tv_sec;         /* seconds */
  suseconds_t tv_usec;        /* microseconds */
};
struct itimerval timer;
timer.it_value.tv_sec = 10;
timer.it_value.tv_usec = 10;
timer.it_interval = timer.it_value;
setitimer(ITIMER_REAL, &timer, NULL);
while(1);
```

A program can set three different types of timers with setitimer:

- If the timer code is ITIMER_REAL, the process is sent a SIGALRM signal after the specified wall-clock time has elapsed.
- If the timer code is ITIMER_VIRTUAL, the process is sent a SIGVTALRM signal after the process has executed for the specified time. Time in which the process is not executing (that is, when the kernel or another process is running) is not counted.

- If the timer code is ITIMER_PROF, the process is sent a SIGPROF signal when the specified time has elapsed either during the process's own execution or the execution of a system call on behalf of the process.

2. Write a program to print the system resource limits. Use getrlimit system call.

man 2 getrlimit

```
struct rlimit {
      rlim_t rlim_cur;  /* Soft limit */
      rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};
```

3. Write a program to set (any one) system resource limit. Use setrlimit system call.

man 2 setrlimit

5. Write a program to print the system limitation of

a. maximum length of the arguments to the exec family of functions.

b. maximum number of simultaneous processes per user id.

c. number of clock ticks (jiffy) per second.

d. maximum number of open files

e. size of a page

f. total number of pages in the physical memory

g. number of currently available pages in the physical memory.

To get these configuration information, use sysconf, man 3 sysconf

6. Write a simple program to create three threads.

man 3 pthread_create

```
void *threadFunc(void *data) {
  static int tid = 1;
  printf(“Thread created”);
  ++tid;
}
main(){
pthread_t tid;
pthread_create(&tid, NULL, threadFunc, NULL); // tid = 1
pthread_create(&tid, NULL, threadFunc, NULL); // tid = 2
pthread_create(&tid, NULL, threadFunc, NULL); // tid = 3
pthread_exit(NULL);
}
```

7. Write a simple program to print the created thread ids.

```
void *threadFunc(void *data) {
pthread_t thread_id = pthread_self();
}
main(){
pthread_t tid;
pthread_create(&tid, NULL, threadFunc, NULL);
pthread_exit(NULL);
}
```

8. Write a separate program using signal system call to catch the following signals.

a. SIGSEGV

b. SIGINT

c. SIGFPE

d. SIGALRM (use alarm system call)

e. SIGALRM (use setitimer system call)

f. SIGVTALRM (use setitimer system call)

g. SIGPROF (use setitimer system call)

man 7 signal

man 2 signal

man 3 raise

man 3 alarm

```
void signal_handler(int signal_no){
  printf("Signal caught");
}
main(){
if(signal(<SIGNAL>, signal_handler) == SIG_ERR){
    perror("Error:");
}
raise(SIGINT);
alarm(1);
}
```

9. Write a program to ignore a SIGINT signal then reset the default action of the SIGINT

signal - Use signal system call.

- **Ctrl + C - SIGINT**
- **Ctrl + \ - SIGQUIT**
- **Ctrl + Z - SIGTSTP**

```
main(){
signal(SIGINT, SIG_IGN); // signal to IGNORE
// may be sleep for few seconds to test
// signal to DEFAULT
// sleep for few seconds to test
}
```

10. Write a separate program using sigaction system call to catch the following signals.

a. SIGSEGV

b. SIGINT

c. SIGFPE

man 2 sigaction

```
void signal_handler(int signal_no){
}
main(){
struct sigaction sa;
sa.sa_handler = signal_handler;
if(sigaction(<SIGNAL>, &sa, NULL) == -1){
    perror("Error:");
}
raise(<SIGNAL>);
}
```

11. Write a program to ignore a SIGINT signal then reset the default action of the SIGINT signal - use sigaction system call.

```
main(){
    struct sigaction sa;
    sa.sa_handler = SIG_IGN;
    sigaction(SIGINT, &sa, NULL); // setting the value using
sigaction
    raise(<SIGNAL>);
    sa.sa_handler = SIG_DFL;
    raise(<SIGNAL>);
}
```

12. Write a program to create an orphan process. Use kill system call to send SIGKILL signal to the parent process from the child process.

man 2 kill

```
fork();
// parent process, put parent in sleep
// child process
// check parent pid before kill , print your ppid
kill(parent_pid, SIGKILL); //getppid()
// sleep for few seconds for kill effect
// check parent pid after kill //getppid()
```

13. Write two programs: first program is waiting to catch SIGSTOP signal, the second program will send the signal (using kill system call). Find out whether the first program is able to catch the signal or not.

```
void signal_handler(int signal_no){
}
main(){
    if(signal(SIGSTOP, signal_handler) == SIG_ERR){
        perror("Error:");
    }
    sleep(30);
}
-------------------------------------------------------
kill(pid, SIGSTOP);
```