Unix System Programming

Chapter 1: The fork() and exec() system calls

The fork()

The fork() system call creates child process under the parent process. The child process starts its execution from the instruction immediately following the fork() call. If there are n fork() calls, 2^n processes will be created.

An important thing to remember about fork() calls is that all the child processes are independent processes, each with its own process ID number. Hence, both parent process as well as the child process can exist concurrently.

Example 1:

```
main() {

     int pid;
     pid=fork();

     if(pid>0) printf("Child process ID: %d",pid);

     else printf("Parent process ID: %d", getppid());
}
```

To the parent process, fork() returns the value of the child's PID.

The moment a call is made to the fork(), a child process is created. The child copy too, gets a copy of the variable "pid", but with a value of 0 by default.

Thus, the variable "pid" of the child process will always be zero.

Sometimes, fork() may not create the child process due to some memory problem. In that case, fork() will return '-1' indicating error.

The getpid() function returns the PID of the process from which it is called. The getppid() function returns the PID if the parent of the process from which it is called.

Orphans and Zombies

Orphans

Example 2:

```
main() {
     int pid;
     pid=fork();

     if (pid==0) { // child part

          printf("I am the child\n");
          printf("My parent's PID: %d\n", getppid());
          printf("My PID: %d\n",getpid());

          sleep(20); // child put to sleep for 20 sec

          printf("I am the child, I have become ORPHAN");
          printf("\nMy PID is: %d",getpid());
          printf("\nMy parent's PID is: %d", getppid());

     } else { // parent's part

          printf("\nI am the Parent & I am dying \n");
          printf("My PID is: %d",getpid());
          printf("\nThe PID of my parent: %d", getppid());

     }
}
```

Here, the child prints the PID of its parent and itself and goes to sleep. In the meanwhile, the parent continues to execute, it prints the PID of it's parent (the shell) and its own PID and terminates.

After 20 seconds, the child wakes up to find that the parent has already terminated. Thus, the child now becomes an orphan.

Unlike in case of humans, the process dispatcher immediately adopts the orphan child and hence after sleep(), the child displays its on PID as before but instead of printing the PID of its original parent, it prints the PID of the process dispatcher as it's parent through the getpid() call.

The ps –l command in UNIX shows 'O' in the second column against the child process, notifying that its original parent has terminated.
Unfortunately, a ps -l command in LINUX will not show 'O' in the second column, because, LINUX does not identify an Orphan process.

The following is the output of the above program when I ran in my PIV machine:

```
I am the child
My parent's PID: 3871
My PID: 3872

I am the Parent & I am dying
My PID is: 3871
The PID of my parent: 3828

[AFTER 20 SEC]

I am the child, I have become ORPHAN
My PID is: 3872
My parent's PID is: 1
```

As we can see, after the parent does, the process dispatcher, with PID=1 adopts the child. Hence, after 20 sec, the child prints the PID of its parent as 1.

Zombies

Zombies are processes that have terminated, but are not removed from the process table. Let us assume that there is a parent process that creates a child. Both of them now have an entry in the process table. Lets further assume that the child process gets terminated well before the parent does. Since the parent process is still in action, the child cannot be removed from the process table. It therefore exists in the twilight zone and thus becomes a zombie.

Lets take an example:

Example 3:

```
main() {
     if (fork()>0) { // parent part

          printf("\nI am the Parent going to sleep\n");
          sleep(20); // parent put to sleep for 20 sec

     } else { // child's part

          printf("\nI am the Child & after my ");
          printf("termination I become a zombie!");
     }
}
```

A command ps -l shows 'Z' in the second column for the child process indicating that it is a zombie.

```
[root@DEVELOPMENT root]$ ps -l

F S  UID PID   PPID  C PRI  NI ADDR  SZ  WCHAN  TTY      TIME      CMD
0 S  500 3828  3826  0 75   0    -  1081 wait4  pts/0    00:00:00 bash
0 T  500 4438  3828  0 75   0    -   335 finish pts/0    00:00:00 a.out
1 Z  500 4439  4438  0 75   0    -     0 do_exi pts/0    00:00:00 a.out
0 R  500 4448  3828  2 75   0    -   781   -    pts/0    00:00:00 ps
```

The above table shows that the child process with PID 4439 is Zombie as was expected.

Kindly note that fork() creates two identical processes but they are totally independent. The two processes share the same variables but each has its own copy of the variables. This can be shown by the example:

Example 4:

```
main() {
     int i=10;
     if(fork()>0) {
          printf("\nThe value of i in parent is %d",i);
     } else {
          i+=10;
          printf("\nThe value of i in child");
          printf(" after incrementation is %d",i);
     }
}
```

The 'exec()' system call

We start with an example to illustrate the exec() function and its differences with fork().

Example 5

ex1.c

```
main() {
     printf("\nBefore exec my PID: %d",getpid());
     printf("\nMy parent's PID: %d",getppid());

     printf("\nExec Starts ...");
     execl("/root/ex2","ex2",(char*)0);
```

```
        printf("\nThis should not print ...\n");
    }
```

ex2.c

```
    main() {
        printf("\nAfter exec, my PID:%d",getpid());
        printf("\nMy Parent PID:%d",getppid());
        printf("\nExec ends\n");
    }
```

Compile the two programs as:

```
[root@DEVELOPMENT root]$ cc –oex1 ex1.c
[root@DEVELOPMENT root]$ cc –oex2 ex2.c
```

That is, the binary files for the two programs should be separately compiled into independent files.

Now, let us analyze this program:

Ex1.c is the main program that calls the ex2 program through the exec() system call. First, it prints the process ID of itself and it's parent. Next, it calls the ex2 program through the execl() system call. This function takes a number of arguments depending upon what parameters we want to pass to the newly called process. The first argument is the path of the ex2 program, i.e., the directory where ex2 resides. The next argument is the name of the program itself, here "ex2". The subsequent arguments are the command line parameters that are passed to ex2. The last argument is always a NULL.

After the call to execl(), the memory space previously occupied by ex1 is overwritten by ex2 program and thus, any code that exists in ex1 program, after the call to execl() never get executed. Here, the printf() function is never called. Instead, the main() of ex2 get executed.

The above program when run in my system gives:

```
    [root@DEVELOPMENT root]$ ./ex1

    Before exec my PID: 3931
    My parent's PID: 3816

    After exec, my PID:3931
    My Parent PID:3816
```

```
Exec ends
```

It is clear from the above output that unlike fork(), execl() does not create an independent process. The PID of the old and the new process is same, indicating that the memory space occupied by the old process is completely overwritten by the new process and the old process no longer has its individual existence.

To better understand execl() call, we take another example:

Example 6

```
ex1.c

main(int argc, char* argv[]) {


     // print PID's here ...

     execl(argv[1],argv[2],argv[3],argv[4],(char*) 0);

     printf("\nThis should not print\n");

}

ex2.c

main(int agc, char* argv[]) {

     printf("\nChild process after exec() call is %s",argv[0]);
     printf(" & its arguments are: %s %s\n",argv[1],argv[2]);
}
```

Run ex1 as:

[root@DEVELOPMENT root]$ ex1 /root/ex2 ex2 Hello World

Here, for ex1 we have:

```
argv[0]="ex1"
argv[1]="/root/ex2"
argv[2]="ex2"
argv[3]="Hello"
argv[4]="World"
```

So, essentially a call to execl() from ex1.c can be written as:

```
execl(“/root/ex2”,”ex2”,”Hello”,”World”,(char*)0);
```

Thus, “Hello” and “World” are the two arguments that are passed to main() of ex2.

In ex2,
```
argv[0]=”ex2” & argv[1]=”Hello” & argv[2]=”World”
```

which gets printed using printf() call.

The execv() call:

We can use ‘execv()’ call instead of execl().

To use execv() in the previous ex1.c program, we modify it as follows:-

Example 7

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

     char *temp[4];

          temp[0]=argv[2];
          temp[1]=argv[3];
          temp[2]=argv[4];
          temp[3]=(char*)0;

          // print PID’s here ...

          execv(argv[1],temp);

          printf(“This should not print \n”);
}
```

Thus, instead of separately specifying the called program along with it’s command line parameters, we put them in an array and specify the name of the array instead. Thus, we prevent hard coding of passed parameters. This brings flexibility in specifying number of arguments to called program as well as run time adjustments.

Chapter 2: Interprocess Communication

In this chapter, we will address the following topics sequentially:-

Signal Passing and Signal trapping
Message Passing
Semaphores

Section A: Signal Passing and Signal Trapping

In any operating system, the different processes need to communicate with each other and processes in general, do not function in isolation. We human beings use different facial expressions as signals to communicate different thoughts, moods etc.

In Unix too, similarly, processes can send each other signals and communicate with each other. For example, say process A has finished processing some data, it may send some signals to process B. Process B will also receive the output of the processed data from Process A. based on this output, process B has to decide what response to give.

In Unix, we use the signal() system call to either trap system generated signals at specified occurrence of events or even to communicate between two user programs.

Let us say for example that we want to trap the 'DEL' key. Each time this key is pressed while a program is run, instead of termination of he current running program, we want that some other user defined function gets executed.
We write the program as follows:-

Example 1:

```
#include <signal.h>

main() {

     printf("Press DEL key \n");
     signal(SIGINT, abc);

     for(;;) // loop forever
}

void abc() {
     printf("\nYou pressed the DEL key \n");
}
```

The first parameter passed to signal() function is te SIGNAL that we want to trap, in this case the signal that kernel generates when DEL key is pressed. SIGINT is the signal

generated and is defined in signal.h. The second argument is the name of the function we want to invoke when the specified signal is generated, here the function abc().

Kindly note that pressing the DEL key twice DOES terminate the program. This is because, the first time DEL is pressed, function abc() is executed and immediately cleared from memory. Subsequent pressing of the DEL key can't find the function abc() in memory and hence the default code is executed (SIG_DFL) and this terminated the program.
To keep the function abc() in memory we modify it as follows:-

```
void abc()
{
          printf("\nYou pressed the DEL key \n");
          signal(SIGINT,abc);
}
```

A recursive signal() call loads abc() back in memory.

What if we wanted this program to ignore the DEL key altogether? Simple! We call the signal() function from main as follows:

```
signal(SIGINT,SIG_IGN)
```

SIG_IGN is a symbolic constant passed to signal() function that instructs the current process to ignore the SIGINT signal.

What if we wanted to write our own custom error handler when any illegal instruction is executed (such as division by zero)? The signal that is generated by the kernel when an illegal operation takes place is the SIGILL signal (defined in signal.h) and we can trap this signal to write our own errorhandler.

Example 2:

```
void main() {
     int I=0;j=50;
     signal(SIGILL,errorhandler);
     j=j/I;    // division by zero, illegal operation
}
void errorhandler(int signalno) {
     // handle error with signal number signalno
}
```

An Important Note:

If you are using Linux (RedHat 6/7/8/9) then this operating system DOES NOT support use of the DEL key for program termination (& generation of SIGINT signal). Instead, use the 'ctrl-c' key to generate SIGINT signal trap.

Further, for illegal floating point exception handling, trap and use the SIGFPE signal instead.

See the Appendix for a full list of all the available signals under Linux.

Process Termination

A process, when terminates, sends a signal to its parent. This parent, if shell, on receiving the signal proceeds the delete the entry for the corresponding process from the process table. The signal sent is SIGCLD.
Even if a forked child dies, it intimates the parent by sending the same signal. Let us illustrate this by an example.

Example 3:

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

void abc();
int pid,i;

main() {

     pid=fork();
     if(pid==0) sleep(1);
     else {
          signal (SIGCLD,abc);
          for(i=0;i<1000;i++);
          printf("\nParent exiting!\n");
     }
}

void abc() {
     printf("\nValue of i is %d",i);
     printf("\nChild Died!\n");
     getchar();
}
```

This program first forks a child and then puts the child for sleep into 1 second. In the meanwhile, the parent continues to execute the for loop. When the child terminates, the child send a SIGCLD signal to the parent. The parent, on receiving this signal calls the abc() function which prints the value of the counter i.

Signals passed between two processes

All the while we have seen communication between the kernel and a user process. The SIGINT was the signal generated by kernel when we press the DEL key. In our program we trapped this signal to invoke our own user defined function abc().

However, signals can also be passed between two user processes as well. The system call kill() is used to send a signal to a process and the receiving process uses the usual signal() call to trap this signal.

The kill() system call takes two arguments. The first argument is the process-id of the process to which we want to send the signal. The second argument is the signal itself defined in signal.h

So, we have a process that sends a signal and a process that receives the signal. Depending upon the signal sent, the receiving process proceeds to take some action.

Normally, this type of communication takes place between the parent and the child since the pid of the parent is known to the child and vice-versa. We take an example:-

Example 4:

```
#include<signal.h>
void abc();

main() {

     int pid;
     pid=fork();
     if(pid==0) {        // child side
          signal(SIGINT,abc);
               // child on receiving signal SIGINT
               // invokes function abc
          sleep(2);
     } else {       // parent side
          kill(pid,SIGINT);
               // sends signal SIGINT to child
          sleep(5);
          printf("Parent Exiting\n");
     }
}

void abc() {
     printf("Signal received by child!!\n");
}
```

Here, we use the predefined SIGINT signal, which is infact, mapped to the DEL key. However, there are two user programmable signals that are not mapped to any KEY and can be effectively used to send signals across processes and invoke user defined operations. They are SIGUSR1 and SIGUSR2 that have values 16 & 17 respectively. For all programming purposes, we must use these two signals and keep the predefined signals like SIGINT reserved for their specific purpose for which they are defined.

Section B: Message Passing Across Processes

Unlike signals, which have predefined values and are all defined in signal.h, messages are like text conversation between any two processes. Signals are generated in response to an event (such as pressing of the DEL key). Messages on the other hand are exchanged between two processes which need to converse with each other. All messages are user defined and does not have any predefined values.

Before we can exchange messages between two processes, we need to create a message queue where the messages will be queued up for delivery. This is the place from where the receiving process can obtain the sent messages. A message queue is more like a mailbox, or like the inbox of our email accounts. Mails sent by different people are queued up in our mailbox until view the emails and decide to delete them one after another.

Message queues are created using the msgget() system call. It takes two arguments; the name of the queue, also known as the key and a flag. The flag can have the following values:

IPC_CREAT which instructs to create the message queue if it does not exist. If it does exist with the specified key value, then this keyword is ignored.

IPC_EXCL which creates the message queue in exclusive mode. That means when it is OR'ed with IPC_CREAT, it forces an error message when the queue with the specified name (key value) already exists.

Along with any one or both these constants, we also specify the queue permission in the usual 'ugo' notation (for example 644 means rw- r-- r-- permission). The permission specifier is ORed with the existing flag constants. Note that an execute permission for a message queue does not make sense since it is not a piece of executable code.

Let is illustrate what we have said by a simple example:-

Example 5:

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

main() {
     int msqid;
     key_t key=15;
     msqid=msgget(key,IPC_CREAT|0644);

     if (msqid<0)
          perror("msgget failed!");
     else
          printf("\nMessage queue created successfully
     with key %d \n",msqid);
}
```

Here, we create a new message queue with the msgget() system call. If successful, it returns the ID of the newly created message queue; otherwise it returns a value less that zero. The key value is a key_t type variable. The structure is defined in ipc.h. The permission 0644 of the created message queue is ORed with the constant IPC_CREAT to form the second argument to the msgget() function.

It may be noted here that once we create a message queue with a specified permission, that queue may be accessed later by using the same permission only. Using a different permission results in an error. However, inn most cases, we may not know the permission with which we first created the queue. In those cases, this error may be avoided by Oring the PIC_CREAT with a 0 (zero) while calling msgget() on an already existing queue.

For example: msqid=msgget(key,IPC_CREAT|0);

Once we create message queues, it is possible for us to view the existing queues by issuing the command:

[root@domain root] # ipcs -q

from the command console. This gives an output of the form:-

```
------ Message Queues --------
key      msqid   owner      perms      used-bytes   messages
0x0000000f 0     root       644        0            0
```

If there is a function to create a queue, there must be one to destroy a queue as well. The msgctl() call does just that. It takes three parameters, the queue identifier, the operation to be done with the queue; in our case it is IPC_RMID (remove queue). The last parameter must be zero (0) if we want to destroy a message queue.

msgctl(msqid, IPC_RMID,0)

Kindly note here that msgctl() call does more that just deleting the queue. We will see more of this function later.

Sending a Message:

We will take up this topic with an example.

Example 6:

send.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
```

```
main() {
     int qid;

     struct {
          long mtype; // message priority
          char mtext[25]; // the text of the message
     } message;

     // the above is the message data structure which
     // contains the message to be sent in the mtext
     // variable

     qid=msgget((key_t) 10, IPC_CREAT|0666);
     // creating the message queue with permission 666

     if(qid==-1) {
          perror("msgget failed!");
          exit(1);
     }

     strcpy(message.mtext,"Good Morning World!!\n");

     // initializing the message structure with the
     // text message we want to send!

     message.mtype=1; // setting priority to 1

     // ** we now send this message using the msgsnd()
     // ** system call **

     if(msgsnd(qid,&message,strlen(message.mtext),0)==-1) {
          perror("msgsnd failed!\n");
          exit(1);
     } else printf("\nMessage successfully sent\n");
}
```

What this program does is at first it creates a message queue with read/write permissions. It then initializes the message data structure with the custom message "Good Morning World!!" and writes this message to the created message queue with priority 1.

The primary function used here in sending the messages is the msgsnd() system call that takes four arguments. The first three arguments is the message queue identifier where to send the message, the starting address of the message data structure and the length of the actual message string.

The last argument needs some explanation. Processes normally wait when the message queue becomes full until there arises enough room for the message to be stored in the queue. Let us imagine that we send so many messages to the queue that the queue

becomes completely filled up and can accept no more incoming messages. In this case, a 0 (zero) as the last argument to msgsnd() will make our program wait until enough room is created in the queue so that the message can be sent. But specifying a IPC_NOWAIT constant as the last argument, we can have the process terminate immediately if there is no more room left.

msgsnd(qid,&message,strlen(message.mtext),IPC_NOWAIT)

Receiving a Message:

There is no point in sending a message that will never ever be received. So, our next objective is to write a program that will receive the message sent by our previous program.

Example 7:

receive.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

main() {
     int qid;
     struct {
          long mtype;
          char mtext[25];
     } buff;

     qid=msgget((key_t)10,IPC_CREAT|0666);

     if(qid==-1) {
          perror("msgget failed!\n");
          exit(1);
     }

     // now receiving the message already sent ...

     if (msgrcv(qid,&buff,21,1,IPC_NOWAIT)==-1) {
          perror("msgrcv failed!\n");
          exit(1);
     }

     printf("message received is %s \n",buff.mtext);
}
```

The main function that receives the message is the msgrcv() function. The first three arguments of this function is same as the previous msgsnd() function. A 0 (zero) as the last argument of this function signifies that our program would wait if the message queue is empty until a single message arrives at the queue with THE SAME PERMISSIONS AS THE ONE IT IS WAITING FOR. We could also specify the last argument as IPC_NOWAIT in which case, our program would not wait if the queue was empty.

The fourth argument is the priority of the message the process seeks to receive. Only those messages that are sent with the matching priority number are accepted. Thus, we can filter out only those messages that have a desired priority number, for acceptance and possibly display on the screen.

One important thing to note in the above program is the length of the message specified(21) as the third argument. This number is hard coded. If this value is less than the length of the message sent, then there would be no truncatation; instead we would get an error message:

```
msgrcv failed: Arg list too long
```

So, how do we avoid this error if we do not know the length of the text message beforehand?
The solution is to OR the last argument of the msgrcv() function with the constant MSG_NOERROR. This will make our program ignore any length that was overshot and instead print only the defined number of characters.

```
msgrcv(qid,&buff,15,1,IPC_NOWAIT|MSG_NOERROR)
```

If we now run the previous program with this modification, we get the output as :-

“message received is Good Morning” with the word “World!!” truncated.

After we run the send.c program, an ipcs -q command will reveal that the message has indeed been sent to the designated queue and is waiting to be received:

```
root@DEVELOPMENT root]$ ipcs -q

------ Message Queues --------
key      msqid   owner      perms      used-bytes   messages
0x0000000a 32769  root       666        21           1
```

The size of the message is shown to be 21 bytes which is exactly the length of the string "Good Morning World!!\n"

Now we run send.c another time without running receive.c and issue the same command.

```
[root@DEVELOPMENT root]$ ipcs -q

------ Message Queues --------
key      msqid   owner      perms      used-bytes   messages
0x0000000a 32769  root       666        42           2
```

We see that now two messages are waiting in the queue, the messages that were sent by running the send.c program twice. The used number of bytes is exactly twice the size of individual messages, as was expected.

Now let us run the receive.c program THRICE. Each time after running the program, we issue the ipcs -q command and find that, number of messages waiting reduce to 1 and finally zero, showing that the receive.c program has eaten up the stored messages. Since, before running the receive.c program for the third time, there are no waiting messages in the queue, we get the error message in the third run:

```
msgrcv failed!
: No message of desired type
```

Assignment:

Write a program which will accept a string as input from the command console and send it as a message to the receiver program. The receiver program upon receiving the message from the sender will display the received message as well as send an acknowledgment to the sender program. The sender program will then display “Acknowledgment received from receiver” and then will wait for the next user input from the console.

Section C: Semaphores

Semaphores are data structures that help to synchronize several processes that access a common resource. Semaphores can be used in any programs that need synchronization and can be created similar to the creation of message queues. In this section, we will see some basic operations on semaphores, such as semaphore creation, deletion, setting its value, deleting its value etc.

The Unix command ipcs –s issued on the console enlists the available semaphores and their properties.

```
[root@DEVELOPMENT root]$ ipcs -s
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 163843     root        666        4
0x0000001a 1048606    root        666        4
```

Unlike queue, semaphores are created in sets. That is, at the most we can have 10 sets of semaphores, each set consisting of a maximum of 25 semaphores.

Semaphore Creation:

If msgget() could create the message queue, for semaphore we have the semget() call tat creates the semaphores. Semget() is used almost exactly like the msgget() call. We will show that with the help if an example:

Example 8

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

main() {
     int semid,nsemset,nsem,flag,key;

     nsem=4; // number of semaphores in each set
     flag=0666|IPC_CREAT; // semaphore permission 666

     for(nsemset=0; ;nsemset++) {
          key=(key_t)nsemset;
          semid=semget(key,nsem,flag);
          if(semid>0) {
               printf("\nSemaphore Created with ID:");
               printf("%d\n",semid);
          } else {
               printf("Maximum num of semaphore set:");
```

```
                printf(" %d\n",nsemset);
                exit(0);
            }
        }
    }
```

This program creates maximum number of possible semaphore sets (10), each set has 4 semaphores. The semget() function takes three arguments, the key or the name of the semaphore set, the number of semaphores to create in that set (here 4) and a flag variable with the permission (666) with IPC_CREAT constant ORed.

Please note that each of the semaphores will have a unique key and all the semaphores belonging to a particular set will have the same key value.

A maximum number of 25 semaphores are allowed for the creation within a single set.

Semaphore Destruction

All the semaphores belonging to a specified set (thus having the same key value) can be killed using the semctl() call.

```
semctl(semid,0,IPC_RMID,0)
```

where the first argument is the semaphore key (a set of semaphores with the same key value), the third argument is the IPC_RMID constant, the second and fourth arguments must be zero. We will explore more of these two arguments when we discuss setting and retrieval of semaphore values.
Please note that when one user or process creates a semaphore with a key value, say 1, another user or process may attempt to create the same semaphore with the same key value. Unless we specify that this semaphore is to be created in the exclusive mode, the second user will not get an error message. All he will be returned the same ID.
However, if we do want an error to be returned, we can always create the semaphore in the exclusive mode by Oring the flag with the IPC_EXCL constant.

For example,

```
semid=semget(key,1,IPC_CREAT|0666|IPC_EXCL)
```

Further, lets assume that the first process created a semaphore with 0666 permission (read-alter-read-alter-read-alter). Now, any attempt by the next process/user would have to be in a permission mode which is the SAME OR IS A SUBSET of the initial mode, for example 0644.

An example will illustrate this fact:-

Example 9

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

main() {
     int semid, key,flag,nsem;
     key=(key_t) 0x30;

     flag=PC_CREAT|0644;
     // first give the permission 644

     nsem=1;

     semid=semget(key,nsem,flag);

     if(semid>0){
          printf("Created semaphore with");
          printf("ID %d\n",semid);
     } else perror("Error in 1st semget\n");

     flag=IPC_CREAT|0666;
     // now changing the permission to 666
     // whch is a superset of 644, the previous
     //permision

     semid=semget(key,nsem,flag);

     if(semid>0){
          printf("Created semaphore with");
          printf("ID %d\n",semid);
     } else perror("Error in 2nd semget\n");
}
```

Here, call to second semget() will result in an error, because it is called with a permission 666 which is a superset of the permission with which the semaphore ws originally created. (644).

Moreover, if we first create a certain number of semaphores in a set and then try to create it once more with another different number for the same set, the second number being greater than the original number of semaphores in the set, then we would get an error in our second attempt. Another call to the semget() will be successful only when in the second call we specify a smaller number of semaphores for the set (compared to the number of semaphores with which the set was originally created).

Retrieving and setting semaphore values:

We used the function semctl() to kill a semaphore. The same function can be used to set and retrieve the value of a semaphore.
Setting the value of the semaphore is simple as we illustrate it with an example:-

Example 10

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

main() {
     int semid,retval;

     semid=semget(0x20,1,0666|IPC_CREAT);

     // using the function semctl to set the value
     semctl(semid,0,SETVAL,1);

     // using the same function to retrieve the value
     // of the semaphore ...
     retval=semctl(semid,0,GETVAL,0);

     printf("The value of the semaphore");
     printf(" after setting: %d\n",retval);

     // doing the same procedure once again ...
     semctl(semid,0,SETVAL,2);

     retval=semctl(semid,0,GETVAL,0);

     printf("The value of the semaphore");
     printf(" after setting: %d\n",retval);
}
```

The call semctl(semid,0,SETVAL,0) sets the value of the semaphore with the key semid. But as we had said before, a set of semaphores can have the same key value. Hence we must provide the index value of the individual semaphore within that set whose value we want to set. This index is the second argument of the function. The third argument is the operation we want to perform on the semaphore, here setting the value. The last argument is the value we want to set.

Semaphore value can be retrieved by invoking the call

semctl(semid,0,GETVAL,0)

Here, the arguments are same as the previous case of setting the value, except that the last argument must be zero(0). This function returns the value of the semaphore with index 0 and having the key value semid.

What if we wanted to get the pid of the process that set the value of the semaphore? Take this example:

Example 11

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<errno.h>

main() {
     int semid,retid;

     // creating the semaphore
     semid=semget(0x20,1,0666|IPC_CREAT);
     // setting value of semaphore
     semctl(semid,0,SETVAL,1);

     // getting the PID of the process that set
     // the semaphore
     retval=semctl(semid,0,GETPID,0);

     printf("PID returned by semctl() is %d & ",retval);
     printf("the actual PID is %d\n",getpid());
}
```

Here, we first set the value of the semaphore and then use GETPID constant as argument to `semctl()` to retrieve the PID of the process that had set the value of the semaphore. In our case, this is same as the current process of the program and hence `retval` and the value returned by `getpid()` should be same.

This concludes our discussion on semaphores.

Reference:

The 'C' Odyssey; Unix – The open Boundless C by Vijay Mukhi; BPB Publications.

Appendix

Available standard kernel signals in Linux (RedHat 6/7/8/9)

Linux supports both POSIX reliable signals (hereinafter "standard signals") and POSIX real-time signals.

Standard Signals

Linux supports the standard signals listed below. Several signal numbers are architecture dependent, as indicated in the "Value" column. (Where three values are given, the first one is usually valid for alpha and sparc, the middle one for i386, ppc and sh, and the last one for mips. A - denotes that a signal is absent on the corresponding architecture.)
The entries in the "Action" column of the table specify the default action for the signal, as follows:

Term
: Default action is to terminate the process.

Ign
: Default action is to ignore the signal.

Core
: Default action is to terminate the process and dump core.

Stop
: Default action is to stop the process.

First the signals described in the original POSIX.1 standard.

Signal	Value	Action	Comment
			or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from *abort*(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from *alarm*(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty

SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

Next the signals not in the POSIX.1 standard but described in SUSv2 and SUSv3 / POSIX 1003.1-2001.

Signal	Value	Action	Comment
SIGPOLL		Term	Pollable event (Sys V). Synonym of SIGIO
SIGPROF	27,27,29	Term	Profiling timer expired
SIGSYS	12,-,12	Core	Bad argument to routine (SVID)
SIGTRAP	5	Core	Trace/breakpoint trap
SIGURG	16,23,21	Ign	Urgent condition on socket (4.2 BSD)
SIGVTALRM	26,26,28	Term	Virtual alarm clock (4.2 BSD)
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2 BSD)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2 BSD)

Up to and including Linux 2.2, the default behaviour for **SIGSYS**, **SIGXCPU**, **SIGXFSZ**, and (on architectures other than SPARC and MIPS) **SIGBUS** was to terminate the process (without a core dump). (On some other Unices the default action for **SIGXCPU** and **SIGXFSZ** is to terminate the process without a core dump.) Linux 2.4 conforms to the POSIX 1003.1-2001 requirements for these signals, terminating the process with a core dump.

Next various other signals.

Signal	Value	Action	Comment
SIGEMT	7,-,7	Term	
SIGSTKFLT	-,16,-	Term	Stack fault on coprocessor (unused)
SIGIO	23,29,22	Term	I/O now possible (4.2 BSD)
SIGCLD	-,-,18	Ign	A synonym for SIGCHLD
SIGPWR	29,30,19	Term	Power failure (System V)
SIGINFO	29,-,-		A synonym for SIGPWR
SIGLOST	-,-,-	Term	File lock lost
SIGWINCH	28,28,20	Ign	Window resize signal (4.3 BSD, Sun)
SIGUNUSED	-,31,-	Term	Unused signal (will be SIGSYS)

(Signal 29 is **SIGINFO** / **SIGPWR** on an alpha but **SIGLOST** on a sparc.)

SIGEMT is not specified in POSIX 1003.1-2001, but neverthless appears on most other Unices, where its default action is typically to terminate the process with a core dump.
SIGPWR (which is not specified in POSIX 1003.1-2001) is typically ignored by default on those other Unices where it appears.
SIGIO (which is not specified in POSIX 1003.1-2001) is ignored by default on several other Unices.

Real-time Signals

Linux supports real-time signals as originally defined in the POSIX.4 real-time extensions (and now included in POSIX 1003.1-2001). Linux supports 32 real-time signals, numbered from 32 (**SIGRTMIN**) to 63 (**SIGRTMAX**). (Programs should always refer to real-time signals using notation **SIGRTMIN**+n, since the range of real-time signal numbers varies across Unices.)
Unlike standard signals, real-time signals have no predefined meanings: the entire set of real-time signals can be used for application-defined purposes. (Note, however, that the LinuxThreads implementation uses the first three real-time signals.)
The default action for an unhandled real-time signal is to terminate the receiving process.
Real-time signals are distinguished by the following:

1. Multiple instances of real-time signals can be queued. By contrast, if multiple instances of a standard signal are delivered while that signal is currently blocked, then only one instance is queued.
2. If the signal is sent using **sigqueue**(2), an accompanying value (either an integer or a pointer) can be sent with the signal. If the receiving process establishes a handler for this signal using the **SA_SIGACTION** flag to **sigaction**(2) then it can obtain this data via the *si_value* field of the *siginfo_t* structure passed as the second argument to the handler. Furthermore, the *si_pid* and *si_uid* fields of this structure can be used to obtain the PID and real user ID of the process sending the signal.
3. Real-time signals are delivered in a guaranteed order. Multiple real-time signals of the same type are delivered in the order they were sent. If different real-time signals are sent to a process, they are delivered starting with the lowest-numbered signal. (I.e., low-numbered signals have highest priority.)

If both standard and real-time signals are pending for a process, POSIX leaves it unspecified which is delivered first. Linux, like many other implementations, gives priority to standard signals in this case.
According to POSIX, an implementation should permit at least _POSIX_SIGQUEUE_MAX (32) real-time signals to be queued to a process. However, rather than placing a per-process limit, Linux imposes a system-wide limit on the number of queued real-time signals for all processes. This limit can be viewed (and with privilege) changed via the */proc/sys/kernel/rtsig-max* file. A related file, */proc/sys/kernel/rtsig-max*, can be used to find out how many real-time signals are currently queued.