# Interprocess Communication(IPC) Programs in C in Ubuntu Linux

In this article author began to see how multiple processes may be running on a machine and maybe be controlled (spawned by fork()) by one of our programs. In numerous applications there is clearly a need for these processes to communicate with each exchanging data or control information. There are a few methods which can accomplish this task.

Let us begin with a simple explanation for the term IPC.

An application can contain one or more processes (programs). A process can have a child process by forking. The child process can do its own work by its own exec method.

This is not to be confused with threads.  
Threads are streams of execution within a process. Threads are not available in the standard C library. We have to add a special library in command line. Though, we can have threads in Unix programs, I am concentrating only on processes, child processes and interaction between processes.   
  
Simple fork() and exec() are used for simplest communication. They neither provide any mechanism for communicating with the child process while it was running, nor allow to communicate with the process outside the parent-child relationship.  
  
Because of these reasons interprocess communication(IPC) was introduced. It provides a way to communicate between parent and children, between unrelated process and even between process on different machines.  
===========================================  
  
Interprocess communication(IPC) is the transfer of data among different  processes.  
  
In this article let us discuss five types of IPC.  
1.Pipes  
2.FIFO-also known as Named pipes   
3.Shared Memory  
4.Message Queue  
5.Semaphore  
(Note: For all the IPCs the header file <sys/ipc.h> should be added)

**PIPES:**  
Pipes are used for unidirectional communication of related process only. The main thing you have to note here is the limitation of the data to be passed.  
Yes, the capacity of data to be sent through pipes are limited. As like standard communication the output of one pipe can be given as an input of another pipe. It is not necessary to free them after the process has been finished.  
  
Then, question arises here...Do we need to create pipes for child process separetly? The answer is, the child process will automatically inherits it.  
The process of sending and receiving the data are done in a linear fashion. The pipes can be either blocked or unblocked.

**FIFO(NAMED PIPES):**

You may now have a question, what should I do if I am in need to communicate with unrelated process?  
  
The solution to this problem is given by means of named pipes. The communication can be done between unrelated processes. Moreover, it can be bi-directional as well.  
One more difference between pipes and named pipes is that the capacity of data i.e. the amount of data flow through a pipe during execution.

**SHARED MEMORY:**

As the name implies, the same area of the physical memory is shared by one or more processes. This memory segment is mapped into virtual memory spaces.

In shared memory two or more process can access the same memory segment. Let us consider a scenario, there are three process p1, p2 and p3. Processes p1 and p2 are accessing the segment in read mode, and p3 in write mode. If p3 makes any changes to the data, then it is automatically updated for the process p1 and p2 also.   
  
Then who would synchronize the processes? Kernel does not take the job of synchronizing, user himself have to synchronize them. The better solution is to use semaphore because it is the efficient way to manage the accessing of resources, here we can mean the process as resource i.e. which process has to get the segment.   
  
The disadvantage of shared memory is it does not benefit from the protection that the operating system normally provides.  
Then what is the advantage of using shared memory? Yes, Of course it has its own advantage, which differentiates it from other forms of IPCs. It is that, Shared memory provides fastest mechanism of communication than the others. The process can read or write without using any system call.

**MESSAGE QUEUE:**

The message queue also provides the way for unrelated processes to communicate with each other.Comparing with other IPC mechanisms this provides the easiest way of communication than others. Instead of sending data one by one, it sends block of data. This is the difference between the pipes and message queue. The difference between message queue and named pipes is that the messages in the queue can be prioritized which cannot be done in the later.  
  
As message queues are attached with the kernel, whenever a message is sent or received it has to visit the kernel, which makes the communication slow. Because of this, the communication process is little bit slower than the shared memory.  
  
As it is a queue, what about the capacity of data? Exactly, we have to concentrate this point too. The capacity of message queue is limited as in pipes.  
  
Next to this, does user have to know the current process working in the program? But unfortunately message queue fails to know which is the current process.  
  
Then you may ask what is the real advantage of making use of message queue. It avoids the problem of blocking and synchronization in named pipes by sending messages.

**SEMAPHORE:**

In a multi-user or multiprocessing systems, there is a problem to have an exclusive access over the shared resource. But it is difficult to write code to ensure the exclusive access to a particular resource.  
  
Then what about setting flags? We can create the files using the O\_EXCL flag, which allows the new process to obtain the token. It also had problem. It suits only simple problems. You may ask why we can’t create a token, which guarantees the access to only one process.  
  
Dutch scientist, Dijkstra introduced the concept of the semaphore as the solution for the problem.Semaphore is a counter variable, which provides the exclusive access to the shared resource for multiple processes.  
  
To get the shared resource, the process has to do the following:  
1.) First you have to test the semphore value, which controls the shared resource.  
2.) If the value is positive, the process can make use of the resource. In this, the semaphore value is decremented by 1 to indicate that one unit of the resources is used.  
3.) If the value is 0,the process goes to sleep. It wakes up only if the semaphore value becomes greater than 0.The semaphore value will be incremented when the process using the resources releases it.   
   
So far I gave a brief introduction about the IPC mechanisms. Now let us have the detailed explanation of how to implement the IPC, ie. The functions and its usage of each mechanism are discussed in detail

1. PIPES:  
A pipe is a serial communication device (i.e., the data is read in the order in which it was written), which allows a unidirectional communication. The data written to end is read back from the other end.  
  
The pipe is mainly used to communicate between two threads in a single process or between parent and child process. Pipes can only connect the related process. In shell, the symbol | can be used to create a pipe.  
  
In pipes the capacity of data is limited. (i.e.) If the writing process  is faster than the reading process which consumes the data, the pipe cannot store the data. In this situation the writer process will block until more capacity becomes available. Also if the reading process tries to read data when there is no data to read, it will be blocked until the data becomes available. By this, pipes automatically synchronize the two process.

**Creating Pipes:**  
The pipe() function provides a means of passing data between two programs and also allows to read and write the data.

#include<unistd.h>  
int pipe(int file\_descriptor[2]);  
  
pipe()function is passed with an array of file descriptors. It will fill the array with new file descriptors and returns zero. On error, returns -1 and sets the errno to indicate the reason of failure.The file descriptors are connected in a way that is data written to file\_ descriptor[1] can be read back from the file\_descriptor[0].

**(Note: As this uses file descriptors and not the file streams, we must use read and write system calls to access the data.)**  
  
**Pipe processing:**  
The process of passing data between two programs can be done with the help of popen() and pclose() functions.  
  
#include<stdio.h>  
FILE \*popen(const char \*command ,    
const char \*open-mode);  
int pclose(FILE \*stream\_to\_close);  
  
popen():  
The popen function allows a program to invoke another program as a new process and either write the data to it or to read from it. The parameter command is the name of the program to run. The open\_mode parameter specifies in which mode it is to be invoked, it can be only either "r" or "w". On failure popen() returns a NULL pointer. **If you want to perform bi-directional communication you have to use two pipes.**  
  
pclose():  
By using pclose(), we can close the filestream associated with popen() after the process started by it has been finished. The pclose() will return the exit code of the process, which is to be closed. If the process was already executed a wait statement before calling pclose, the exit status will be lost because the process has been finished. After closing the filestream, pclose() will wait for the child process to terminate.

Pipes used as standard input and output:   
We can invoke the standard programs, ones that don’t expect a file descriptor as a parameter.  
  
#include<unistd.h>  
int dup(int file\_descriptor);  
int dup2(int file\_descriptor\_1,  
int file\_descriptor\_2);

The purpose of dup call is to open a new file descriptor, which will refer to the same file as an existing file descriptor. In case of dup, the value of the new file descriptor is the lowest number available. In dup2 it is same as, or the first available descriptor greater than the parameter file\_descriptor\_2.  
  
We can pass data between process by first closing the file descriptor 0 and call is made to dup. By this the new file descriptor will have the number 0.As the new descriptor is the duplicate of an existing one, standard input is changed to have the access. So we have created two file descriptors for same file or pipe, one of them will be the standard input.

(Note: The same operation can be performed by using the fcntl() function. But compared to this dup and dup2 are more efficient)

//pipes.c  
#include<unistd.h>  
#include<stdlib.h>  
#include<stdio.h>  
#include<string.h>

int main()  
{  
int data\_processed;  
int file\_pipes[2];  
const char some\_data[]= "123";  
pid\_t fork\_result;  
  
if(pipe(file\_pipes)==0)  
{  
fork\_result=fork();  
if(fork\_result==(pid\_t)-1)  
{  
fprintf(stderr,"fork failure");  
exit(EXIT\_FAILURE);  
}

   if(fork\_result==(pid\_t)0)  
{  
close(0);  
dup(file\_pipes[0]);  
close(file\_pipes[0]);  
close(file\_pipes[1]);  
execlp("od","od","-c",(char \*)0);  
exit(EXIT\_FAILURE);  
}  
else  
{  
close(file\_pipes[0]);  
data\_processed=write(file\_pipes[1],  
some\_data,strlen(some\_data));  
close(file\_pipes[1]);  
printf("%d -wrote %d bytes\n",(int)getpid(),data\_processed);  
}  
}  
exit(EXIT\_SUCCESS);  
}  
  
The program creates a pipe and then forks. Now both parent and child process will have its own file descriptors for reading and writing. Therefore totally there are four file descriptors.  
  
The child process will close its standard input with close(0) and calls dup(file\_pipes[0]). This will duplicate the file descriptor associated with the read end. Then child closes its original file descriptor. As child will never write, it also closes the write file descriptor,  
file\_pipes[1]. Now there is only one file descriptor 0 associated with the pipe that is standard input. Next, child uses the exec to invoke any program that reads standard input. The od command will wait for the data to be available from the user terminal.  
  
Since the parent never read the pipe, it starts by closing the read end that is file\_pipe[0]. When writing process of data has been finished, the write end of the parent is closed and exited. As there are no file descriptor open to write to pipe, the od command will be able to read the three bytes written to pipe, meanwhile the reading process will return 0 bytes indicating the end of the file.

2. FIFO:

Above we have discussed how to perform communication between two related process. Now, let us discuss how to communicate between the unrelated process.  
  
A First-in, first-out(FIFO) file is a pipe that has a name in the filesystem. It is also called as named pipes.

Creation of FIFO:   
We can create a FIFO from the command line and within a program.  
  
To create from command line we can use either mknod or mkfifo commands.  
$ mknod filename p  
$ mkfifo filename  
  
(Note: The mknod command is available only n older versions, you can make use of mkfifo in new versions.)   
  
To create FIFO within the program we can use two system calls. They are,  
#include<sys/types.h>  
#include<sys/stat.h>  
  
int mkfifo(const char \*filename,mode\_t mode);  
int mknod(const char \*filename,mode\_t mode|S\_IFIFO,(dev\_t) 0);  
  
If we want to use the mknod function we have to use ORing process of fileaccess mode with S\_IFIFO and the dev\_t value of 0.Instead of using this we can use the simple mkfifo function.

Accessing FIFO:  
Let us first discuss how to access FIFO in command line using file commmands. The useful feature of named pipes is, as they appear in the file system, we can use them in commands.

We can read from the FIFO(empty)  
$ cat < /tmp/my\_fifo  
Now, let us write to the FIFO.  
$ echo "Simple!!!" > /tmp/my\_fifo  
(Note: These two commands should be executed in different terminals because first command will be waiting for some data to appear in the FIFO.)  
   
FIFO can also be accessed as like a file in the program using low-level I/O functions or C library I/O functions.  
  
The only difference between opening a regular file and FIFO is the use of open\_flag with the option O\_NONBLOCK. The only restriction is that we can’t open FIFO for reading and writing with O\_RDWR mode.

//fifo1.c

#include <unistd.h>  
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
#include <fcntl.h>  
#include <limits.h>  
#include <sys/types.h>  
#include <sys/stat.h>

#define FIFO\_NAME "/tmp/my\_fifo"  
#define BUFFER\_SIZE PIPE\_BUF  
#define TEN\_MEG (1024 \* 1024 \* 10)

int main()  
{  
int pipe\_fd;  
int res;  
int open\_mode = O\_WRONLY;  
int bytes\_sent = 0;  
char buffer[BUFFER\_SIZE + 1];  
if (access(FIFO\_NAME, F\_OK) == -1) {  
res = mkfifo(FIFO\_NAME, 0777);  
if (res != 0) {  
fprintf(stderr, "Could not create fifo %s\n", FIFO\_NAME);  
exit(EXIT\_FAILURE);  
}  
}  
printf("Process %d opening FIFO O\_WRONLY\n", getpid());  
pipe\_fd = open(FIFO\_NAME, open\_mode);  
printf("Process %d result %d\n", getpid(), pipe\_fd);  
if (pipe\_fd != -1) {  
while(bytes\_sent < TEN\_MEG) {  
res = write(pipe\_fd, buffer, BUFFER\_SIZE);  
if (res == -1) {  
fprintf(stderr, "Write error on pipe\n");  
exit(EXIT\_FAILURE);  
}  
}  
(void)close(pipe\_fd);  
}  
else {  
exit(EXIT\_FAILURE);  
}  
printf("Process %d finished\n", getpid());  
exit(EXIT\_SUCCESS);  
}

//fifo2.c   
#include <unistd.h>  
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
#include <fcntl.h>  
#include <limits.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#define FIFO\_NAME "/tmp/my\_fifo"  
#define BUFFER\_SIZE PIPE\_BUF  
int main()  
{  
int pipe\_fd;  
int res;  
int open\_mode = O\_RDONLY;  
char buffer[BUFFER\_SIZE + 1];  
int bytes\_read = 0;  
memset(buffer, '\0', sizeof(buffer));  
printf("Process %d opening FIFO O\_RDONLY\n", getpid());  
pipe\_fd = open(FIFO\_NAME, open\_mode);  
printf("Process %d result %d\n", getpid(), pipe\_fd);  
if (pipe\_fd != -1) {  
do {  
res = read(pipe\_fd, buffer, BUFFER\_SIZE);  
bytes\_read += res;  
} while (res > 0);  
(void)close(pipe\_fd);  
}  
else {  
exit(EXIT\_FAILURE);  
}  
printf("Process %d finished, %d bytes read\n", getpid(), bytes\_read);  
exit(EXIT\_SUCCESS);  
}

Both fifo1.c and fifo2.c programs use the FIFO in blocking mode.  
  
First fifo1.c is executed .It blocks and waits for reader to open the named pipe. Now writer unblocks and starts writing data to pipe. At the same time, the reader starts reading data from the pipe.  
  
**3. SHARED MEMORY:**

Shared memory is a highly efficient way of data sharing between the running programs. It allows two unrelated processes to access the same logical memory. It is the fastest form of IPC because all processes share the same piece of memory. It also avoids copying data unnecessarily.  
  
As kernel does not synchronize the processes, it should be handled by the user. Semaphore can also be used to synchronize the access to shared memory.

**Usage of shared memory:**  
To use the shared memory, first of all one process should allocate the segment, and then each process desiring to access the segment should attach the segment. After accessing the segment, each process should detach it. It is also necessary to deallocate the segment without fail.  
  
Allocating the shared memory causes virtual pages to be created. It is important to note that allocating the existing segment would not create new pages, but will return the identifier for the existing pages.  
  
All the shared memory segments are allocated as the integral multiples of the system's page size, which is the number of bytes in a page of memory.

**Allocation:**  
A process can allocate a shared memory by using shmget() function.  
  
#include<sys/shm.h>  
int shmget(key\_t key,size\_t size,int shmflg);  
  
The first parameter is the integer key that specifies which segment to create. Unrelated processes can also use the shared memory by specifying the same key value. You can raise a question what would happen if another process also chosen the same fixed key! Obviously it would lead to conflict. There is a solution to this problem; you can use a special constant IPC\_PRIVATE as the key value, which would guarantee creation of new memory segment.  
  
The second parameter specifies the number of bytes in the segment. You have to mention this because the segment is allocated using pages; the number of allocated bytes are rounded up to an integral multiple of the page size.  
  
The third parameter is the flag values, which specifies the options to shmget. The flag values are:  
  
1.) IPC\_CREAT-indicates that a new segment should be created.  
2.) IPC\_EXCL-this is always used with IPC\_CREAT.it causes the shmget to fail if its key value already exists.  
3.) Mode flags -this is the value of 9 bits indicating the permission granted to owner, group and others to access the segment.

**Attachment:**  
When the shared memory segment is created, it could not be accessed by any process. To enable the access it must be attached to the address space of a process. This can be done with the help of shmat() function.  
  
#include<sys/shm.h>  
void \*shmat(int shm\_id,const void \*shm\_addr,int shmflg);  
  
The first parameter is the shm\_id,identifier returned by the shmget.  
The second parameter is the address at which the segment is to be attached to the current process. This will always be NULL pointer so that system can choose the address at which the memory is available.  
The third parameter is the flag value. There are two possible flag values they are:  
1.) SHM\_RND-controls the address at which the segment is attached.  
2.) SHM\_RDONLY-makes the attachment read-only.  
  
If the calls succeed, it will return the address of the attached shared segment.

**Detachment:**  
The shmdt function will detach the shared memory from the current process.

#include<sys/shm.h>  
int shmdt(const void \*shm\_addr);  
It takes the pointer to the address returned by the shmget as the parameter.   On success it returns 0,and -1 on error.

**Controlling and deallocation:**  
The shmctl()will return the information about a shared memory segment and it can also be modified.  
  
#include<sys/shm.h>  
int shmctl(int shm\_id,int command,struct shmid\_ds \*buf);  
  
The second parameter command is the action, which is to be taken. The available values are:  
1.) IPC\_STAT - sets the data in the shmid\_ds structure to reflect the values associated with the shared memory.  
2.) IPC\_SET - sets the values associated with the shared memory to those provided in the shmid\_ds data structure, if the process has permission to do so.  
3.) IPC\_RMID - Deletes the shared memory segment.  
  
The third parameter, buf is a pointer to the structure containing the modes and permissions for the shared memory.

Each shared memory should be explicitly deallocated.

//shmry1.c

#include<unistd.h>  
#include<stdlib.h>  
#include<stdio.h>  
#include<string.h>  
#include<sys/shm.h>

#define TEXT\_SZ 2048  
  
struct shared\_use\_st  
{  
int written\_by\_you;  
char some\_text[TEXT\_SZ];  
};

int main()  
{  
int running = 1;  
void \*shared\_memory = (void \*)0;  
struct shared\_use\_st \*shared\_stuff;  
int shmid;  
srand( (unsigned int)getpid() );  
  
shmid = shmget( (key\_t)1234,  
sizeof(struct shared\_use\_st),  
0666 |IPC\_CREAT );

if (shmid == -1)  
{  
fprintf(stderr, "shmget failed\n");  
exit(EXIT\_FAILURE);  
}  
shared\_memory = shmat(shmid,(void \*)0, 0);  
if (shared\_memory == (void \*)-1)   
{  
fprintf(stderr, "shmat failed\n");  
exit(EXIT\_FAILURE);  
}  
  
printf("Memory Attached at %x\n",  
(int)shared\_memory);

 shared\_stuff = (struct shared\_use\_st \*)  
shared\_memory;  
shared\_stuff->written\_by\_you = 0;  
while(running)  
  
{  
if(shared\_stuff->written\_by\_you)  
{  
  
printf("You Wrote: %s",shared\_stuff->some\_text);  
  
sleep( rand() %4 );  
shared\_stuff->written\_by\_you = 0;  
  
if (strncmp(shared\_stuff->some\_text,"end", 3)== 0)  
{  
running = 0;  
}  
}  
}  
  
  
if (shmdt(shared\_memory) == -1)  
  
{  
fprintf(stderr, "shmdt failed\n");  
exit(EXIT\_FAILURE);  
}  
if (shmctl(shmid, IPC\_RMID, 0) == -1)  
{  
fprintf(stderr, "failed to delete\n");  
exit(EXIT\_FAILURE);  
}

  exit(EXIT\_SUCCESS);

}

//shmry2.c

#include<unistd.h>  
#include<stdlib.h>  
#include<stdio.h>  
#include<string.h>

#include<sys/shm.h>

#define TEXT\_SZ 2048  
struct shared\_use\_st  
{  
int written\_by\_you;  
char some\_text[TEXT\_SZ];  
};  
  
int main()  
{  
int running =1;  
void \*shared\_memory = (void \*)0;  
struct shared\_use\_st \*shared\_stuff;  
char buffer[BUFSIZ];  
int shmid;

shmid =shmget( (key\_t)1234,sizeof(struct shared\_use\_st),0666 | IPC\_CREAT);  
if (shmid == -1)  
{  
fprintf(stderr, "shmget failed\n");  
exit(EXIT\_FAILURE);  
}

shared\_memory=shmat(shmid,(void \*)0, 0);  
if (shared\_memory == (void \*)-1)   
{  
fprintf(stderr, "shmat failed\n");  
exit(EXIT\_FAILURE);  
}  
  
printf("Memory Attached at %x\n", (int) shared\_memory);  
shared\_stuff = (struct shared\_use\_st \*)shared\_memory;  
while(running)  
{  
while(shared\_stuff->written\_by\_you== 1)  
{  
sleep(1);  
printf("waiting for client....\n");  
}  
printf("Enter Some Text: ");  
fgets (buffer, BUFSIZ, stdin);  
strncpy(shared\_stuff->some\_text, buffer,   
TEXT\_SZ);  
shared\_stuff->written\_by\_you = 1;  
if(strncmp(buffer, "end", 3) == 0)  
{  
running = 0;  
}  
}  
if (shmdt(shared\_memory) == -1)  
{  
fprintf(stderr, "shmdt failed\n");  
exit(EXIT\_FAILURE);  
}  
exit(EXIT\_SUCCESS);  
  
}

The shmry1.c program will create the segment using shmget() function and returns the identifier shmid. Then that segment is attached to its address space using shmat() function.  
  
The structure share\_use\_st consists of a flag written\_by\_you is set to 1 when data is available. When it is set, program reads the text, prints it and clears it to show it has read the data. The string end is used to quit from the loop. After this the segment is detached and deleted.  
  
The shmry2.c program gets and attaches to the same memory segment. This is possible with the help of same key value 1234 used in the shmget() function. If the written\_by\_you text is set, the process will wait until the previous process reads it. When the flag is cleared, the data is written and sets the flag. This program too will use the string "end" to terminate. Then the segment is detached.

**4.MESSAGE QUEUE:**  
This is an easy way of passing message between two process. It provides a way of sending a block of data from one process to another. The main advantage of using this is, each block of data is considered to have a type, and a receiving process receives the blocks of data having different type values independently.

**Creation and accessing of a message queue:**  
You can create and access a message queue using the msgget() function.  
  
#include<sys/msg.h>  
int msgget(key\_t key,int msgflg);  
  
The first parameter is the key value, which specifies the particular message queue. The special constant IPC\_PRIVATE will create a private queue. But on some Linux systems the message queue may not actually be private.  
  
The second parameter is the flag value, which takes nine permission flags.

**Adding a message:**  
The msgsnd() function allows to add a message to a message queue.  
#include<sys/msg.h>  
int msgsnd(int msqid,const void \*msg\_ptr,size\_t msg\_sz,int msgflg);  
  
The first parameter is the message queue identifier returned from an msgget function.  
  
The second parameter is the pointer to the message to be sent. The third parameter is the size of the message pointed to by msg\_ptr. The fourth parameter, is the flag value controls what happens if either the current message queue is full or within the limit. On success, the function returns 0 and a copy of the message data has been taken and placed on the message queue, on failure -1 is returned.

**Retrieving a message:**  
The smirch() function retrieves message from the message queue.  
  
#include<sys/msg.h>  
int msgsnd(int msqid,const void \*msg\_ptr,size\_t msg\_sz,long int msgtype ,int msgflg);  
  
The second parameter is a pointer to the message to be received.  
The fourth parameter allows a simple form of reception priority. If its value is 0,the first available message in the queue is retreived. If it is greater than 0,the first message type is retrived. If it is less than 0,the first message that has a type the same a or less than the absolute value of msgtype is retrieved.  
  
On success, msgrcv returns the number on bytes placed in the receive buffer, the message is copied into the user-allocated buffer and the data is deleted from the message queue. It returns -1 on error.

**Controlling the message queue:**  
This is very similar that of control function of shared memory.   
  
#include<sys/msg.h>  
int msgctl(int msgid,int command,struct msqid\_ds \*buf);  
  
The second parameter takes the values as given below:  
  
1.) IPC\_STAT - Sets the data in the msqid\_ds to reflect the values associated with the message queue.  
  
2.) IPC\_SET - If the process has the permission to do so, this sets the values associated with the message queue to those provided in the msgid\_ds data structure.  
  
3.) IPC\_RMID-Deletes the message queue.

(Note: If the message queue is deleted while the process is writing in a msgsnd or msgrcv function, the send or receive function will fail.)

//msgq1.c   
#include<stdlib.h>  
#include<stdio.h>  
#include<string.h>  
#include<errno.h>  
#include<unistd.h>

#include<sys/msg.h>

struct my\_msg\_st  
{  
long int my\_msg\_type;  
char some\_text[BUFSIZ];  
};

int main(){  
int running = 1;  
int msgid;  
struct my\_msg\_st some\_data;  
long int msg\_to\_receive = 0;

msgid = msgget( (key\_t)1234,   
0666 | IPC\_CREAT);  
if (msgid == -1)  
{  
fprintf(stderr, "failed to get:\n");  
exit(EXIT\_FAILURE);  
}

 while (running)  
{  
if(msgrcv(msgid, (void \*)&some\_data,BUFSIZ,msg\_to\_receive,0)  == -1)  
{  
fprintf(stderr, "failedto receive: \n");  
exit(EXIT\_FAILURE);  
}  
printf("You Wrote: %s",some\_data.some\_text);  
if(strncmp(some\_data.some\_text, "end", 3)== 0)  
{  
running = 0;

}  
}  
if (msgctl(msgid, IPC\_RMID, 0) == -1)  
{  
fprintf(stderr, "failed to delete\n");  
exit(EXIT\_FAILURE);  
}  
exit(EXIT\_SUCCESS);  
}  
  
//msgq2.c

#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
#include <errno.h>  
#include <unistd.h>  
#include <sys/msg.h>  
#define  MAX\_TEXT 512

struct my\_msg\_st  
{  
long int my\_msg\_type;  
char some\_text[MAX\_TEXT];  
};  
  
int main()  
{  
int running = 1;  
struct my\_msg\_st some\_data;  
int msgid;  
char buffer[BUFSIZ];

msgid = msgget( (key\_t)1234,0666 | IPC\_CREAT);  
if (msgid == -1)  
{  
fprintf(stderr, "failed to create:\n");  
exit(EXIT\_FAILURE);  
}  
while(running)  
{  
printf("Enter Some Text: ");  
fgets(buffer, BUFSIZ, stdin);  
some\_data.my\_msg\_type = 1;  
strcpy(some\_data.some\_text, buffer);  
  
if(msgsnd(msgid, (void \*)&some\_data, MAX\_TEXT, 0) == -1)  
{  
fprintf(stderr, "msgsnd failed\n");  
exit(EXIT\_FAILURE);  
}  
if(strncmp(buffer, "end", 3) == 0)  
{  
running = 0;  
}  
}  
exit(EXIT\_SUCCESS);  
}

The msgq1.c program will create the message queue using msgget() function. The msgid identifier is returned by the msgget().The message are received from the queue using msgrcv() function until the string "end" is encountered. Then the queue is deleted using msgctl() function.  
  
The msgq2.c program uses the msgsnd() function to send the entered text to the queue.

**5. SEMAPHORE:**

**While we are using threads in our programs in multi-user systems, multiprocessing system, or a combination of two, we may often discover critical sections in the code. This is the section where we have to ensure that a single process has exclusive access to the resource.**  
  
For this purpose the semaphore is used. It allows in managing the access to resource.  
To prevent the problem of one program accessing the shared resource simultaneously, we are in need to generate and use a token which guarantees the access to only one thread of execution in the critical section at a time.  
  
It is counter variable, which takes only the positive numbers and upon which programs can only act atomically. The positive number is the value indicating the number of units of the shared resources are available for sharing.  
  
The common form of semaphore is the binary semaphore, which will control a single resource, and its value is initialized to 0.

**Creation of semaphore:**  
The shmget() function creates a new semaphore or obtains the semaphore key of an existing semaphore.

#include<sys/sem.h>  
int semget(key\_t key,int num\_sems,int sem\_flags);  
  
The first parameter, key, is an integral value used to allow unrelated process to access the same semaphore. The semaphore key is used only by semget. All others use the identifier return by the semget(). There is a special key value IPC\_PRIVATE which allows to create the semaphore and to be accessed only by the creating process.  
  
The second parameter is the number of semaphores required, it is almost always 1.  
The third parameter is the set of flags. The nine bits are the permissions for the semaphore.  
  
On success it will return a positive value which is the identifier used by the other semaphore functions. On error, it returns -1.

Changing the value:  
The function semop() is used for changing the value of the semaphore.

#include<sys/sem.h>  
int semop(int sem\_id,struct sembuf \*sem\_ops,size\_t num-\_sem\_ops);  
  
The first parameter is the shmid is the identifier returned by the semget().  
The second parameter is the pointer to an array of structure. The structure may contain at least the following members:  
struct sembuf{  
short sem\_num;  
short sem\_op;  
short sem\_flg;}  
The first member is the semaphore number, usually 0 unless it is an array of semaphore. The sem\_op is the value by which the semaphore should be changed. Generally it takes -1,which is operation to wait for a semaphore and +1, which is the operation to signal the availability of semaphore.  
  
The third parameter, is the flag which is usually set to SET\_UNDO. If the process terminates without releasing the semaphore, this allows to release it automatically.  
  
**Controlling the semaphore:**  
The semctl() function allows direct control of semaphore information.

#include<sys/sem.h>  
int semctl(int sem\_id,int sem\_num,int command,.../\*union semun arg \*/);  
  
The third parameter is the command, which defines the action to be taken. There are two common values:  
  
1.) SETVAL: Used for initializing a semaphore to a known value.  
2.) IPC\_RMID:Deletes the semaphore identifier.

//sem.c   
#include <unistd.h>  
#include <stdlib.h>  
#include <stdio.h>  
#include <sys/sem.h>  
#include<sys/ipc.h>  
#include<sys/types.h>

union semun  
{  
int val;  
struct semid\_ds \*buf;  
unsigned short \*array;  
};

static void del\_semvalue(void);  
static int set\_semvalue(void);  
static int semaphore\_p(void);  
static int semaphore\_v(void);  
static int sem\_id;  
static int set\_semvalue()  
{  
union semun sem\_union;  
sem\_union.val = 1;  
if (semctl(sem\_id, 0, SETVAL, sem\_union) == -1) return(0);  
return(1);  
}

static void del\_semvalue()  
{  
union semun sem\_union;  
if (semctl(sem\_id, 0, IPC\_RMID, sem\_union) == -1)  
fprintf(stderr, "Failed to delete semaphore\n");  
}

static int semaphore\_p()  
{  
struct sembuf sem\_b;  
sem\_b.sem\_num = 0;  
sem\_b.sem\_op = -1; /\* P() \*/  
sem\_b.sem\_flg = SEM\_UNDO;  
if (semop(sem\_id, &sem\_b, 1) == -1)  
{  
fprintf(stderr, "semaphore\_p failed\n");  
return(0);  
}  
return(1);  
}

static int semaphore\_v()  
{  
struct sembuf sem\_b;  
sem\_b.sem\_num = 0;  
sem\_b.sem\_op = 1; /\* V() \*/  
sem\_b.sem\_flg = SEM\_UNDO;  
if (semop(sem\_id, &sem\_b, 1) == -1) {  
fprintf(stderr, "semaphore\_v failed\n");  
return(0);  
}  
return(1);  
}

int main(int argc, char \*argv[])  
{  
int i;  
int pause\_time;  
char op\_char = 'O';  
srand((unsigned int)getpid());  
sem\_id = semget((key\_t)1234, 1, 0666 | IPC\_CREAT);  
if (argc > 1)  
{  
if (!set\_semvalue())  
{  
fprintf(stderr, "Failed to initialize semaphore\n");  
exit(EXIT\_FAILURE);  
}  
op\_char = 'X';  
sleep(2);  
}

for(i = 0; i < 10; i++)   
{  
if (!semaphore\_p()) exit(EXIT\_FAILURE);  
printf("%c", op\_char);  
fflush(stdout);  
pause\_time = rand() % 3;  
sleep(pause\_time);  
printf("%c", op\_char);fflush(stdout);

if (!semaphore\_v()) exit(EXIT\_FAILURE);  
pause\_time = rand() % 2;  
sleep(pause\_time);  
}  
printf("\n%d - finished\n", getpid());  
if (argc > 1)   
{  
sleep(10);  
del\_semvalue();  
}  
exit(EXIT\_SUCCESS);  
}

The function set\_semvalue() initializes the semaphore using the SETVAL command in semctl() function. But this is to be done before the usage of semaphore.  
  
The function del\_semvalue() is used to delete the semaphore by using the command IPC\_RMID in the semctl() function. The function semaphore\_p() changes the semaphore value to -1, which is used to make the process to wait.  
  
In the function semaphore\_v(),the semop member of the structure sembuf is set to 1.By this the semphore becomes available for the other processes because it is released.  
**Here we have implemented the binary semaphore only**. If you want to extend make the number of semaphores as you need.