Operating System and System Administration
Sri Lanka Institute of Information Technology
Worksheet 8

Purpose
In this lab, You will learn about Intercrosses communication with Message queue.

With System V, AT&T introduced three new forms of IPC facilities (message queues, semaphores, and shared memory). Each IPC *object* has a unique IPC identifier associated with it. When we say ``IPC object'', we are speaking of a single message queue, semaphore set, or shared memory segment. This identifier is used within the kernel to uniquely identify an IPC object. For example, to access a particular shared memory segment, the only item you need is the unique ID value which has been assigned to that segment.
The uniqueness of an identifier is relevant to the *type* of object in question. To illustrate this, assume a numeric identifier of "12345". While there can never be two message queues with this same identifier, there exists the distinct possibility of a message queue and, say, a shared memory segment, which have the same numeric identifier.

IPC Keys

To obtain a unique ID, a *key* must be used. The key must be mutually agreed upon by both client and server processes. This represents the first step in constructing a client/server framework for an application. Often, the ftok() function is used to generate key values for both the client and the server.

LIBRARY FUNCTION: ftok();
PROTOTYPE: key_t ftok (char *pathname, char proj);
RETURNS: new IPC key value if successful
-1 if unsuccessful, errno set to return of stat() call

The ipcs Command

The ipcs command can be used to obtain the status of all System V IPC objects

ipcs -q: Show only message queues
ipcs -s: Show only semaphores
ipcs -m: Show only shared memory
ipcs --help: Additional arguments

The ipcrm Command

The ipcrm command can be used to remove an IPC object from the kernel. While IPC objects can be removed via system calls in user code , the need often arises, especially under development environments, to remove IPC objects manually. Its usage is simple:

ipcrm <msg | sem | shm> <IPC ID>

Simply specify whether the object to be deleted is a message queue (*msg*), a semaphore set (*sem*), or a shared memory segment (*shm*). The IPC ID can be obtained by the ipcs command. You have to specify the type of object, since identifiers are unique among the same type

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. Each message queue (of course) is uniquely identified by an IPC identifier.

Internal and User Data Structures

The key to fully understanding such complex topics as System V IPC is to become intimately familiar with the various internal data structures that reside within the confines of the kernel itself. Direct access to some of these structures is necessary for even the most primitive operations, while others reside at a much lower level

Message buffer

The first structure we'll visit is the msgbuf structure. This particular data structure can be thought of as a *template* for message data. While it is up to the programmer to define structures of this type, it is imperative that you understand that there is actually a structure of type msgbuf. It is declared in linux/msg.h as follows:

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
   long mtype;        /* type of message */
   char mtext[1];     /* message text */ };
```

Kernel msg structure

The kernel stores each message in the queue within the framework of the msg structure. It is defined for us in linux/msg.h as follows:

```
/* one msg structure for each message */
struct msg {
   struct msg *msg_next;   /* next message on queue */
   long  msg_type;
   char *msg_spot;        /* message text address */
   short msg_ts;          /* message text size */
};
```

Kernel msqid_ds structure

Each of the three types of IPC objects has an internal data structure which is maintained by the kernel. For message queues, this is the msqid_ds structure. The kernel creates, stores, and maintains an instance of this structure for every message queue created on the system. It is defined in linux/msg.h as follows:

Kernel ipc_perm structure
The kernel stores permission information for IPC objects in a structure of type ipc_perm. For example, in the internal structure for a message queue described above, the msg_perm member is of this type. It is declared for us in linux/ipc.h as follows:

SYSTEM CALL: msgget()
In order to create a new message queue, or access an existing queue, the msgget() system call is used.

SYSTEM CALL: msgsnd()
Once we have the queue identifier, we can begin performing operations on it. To deliver a message to a queue, you use the msgsnd system call:

SYSTEM CALL: msgctl()
Through the development of the wrapper functions presented earlier, you now have a simple, somewhat elegant approach to creating and utilizing message queues in your applications. Now, we will turn the discussion to directly manipulating the internal structures associated with a given message queue.

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

#define MAX_SEND_SIZE 80

        struct mymsgbuf {
              long mtype;
              char mtext[MAX_SEND_SIZE];
        };

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);
```

```
---------------------------------------------------------------------------------------------
int main(int argc, char *argv[])
{
	key_t key;
	int   msgqueue_id;
	struct mymsgbuf qbuf;

	if(argc == 1)
		usage();

	/* Create unique key via call to ftok() */
	key = ftok(".", 'm');

	/* Open the queue - create if necessary */
	if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
		perror("msgget");
		exit(1);
	}

	switch(tolower(argv[1][0]))
	{
		case 's': send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
					atol(argv[2]), argv[3]);
			  break;
		case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
			  break;
		case 'd': remove_queue(msgqueue_id);
			  break;
		case 'm': change_queue_mode(msgqueue_id, argv[2]);
			  break;

		 default: usage();

	}

	return(0);
}
---------------------------------------------------------------------------------------------
	void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
	{
		/* Send a message to the queue */
		printf("Sending a message ...\n");
		qbuf->mtype = type;
		strcpy(qbuf->mtext, text);

		if((msgsnd(qid, (struct msgbuf *)qbuf,
```

```
                strlen(qbuf->mtext)+1, 0)) ==-1)
            {
                perror("msgsnd");
                exit(1);
            }
        }
----------------------------------------------------------------------------
void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
    /* Read a message from the queue */
    printf("Reading a message ...\n");
    qbuf->mtype = type;
    msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);

    printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
}
----------------------------------------------------------------------------
            void remove_queue(int qid)
            {
                /* Remove the queue */
                msgctl(qid, IPC_RMID, 0);
            }
----------------------------------------------------------------------------
void change_queue_mode(int qid, char *mode)
{
    struct msqid_ds myqueue_ds;

    /* Get current info */
    msgctl(qid, IPC_STAT, &myqueue_ds);

    /* Convert and load the mode */
    sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);

    /* Update the mode */
    msgctl(qid, IPC_SET, &myqueue_ds);
}
----------------------------------------------------------------------------
            void usage(void)
            {
                fprintf(stderr, "A utility for message queues\n");
                fprintf(stderr, "\nUSAGE:  (s)end <type> <messagetext>\n");
                fprintf(stderr, "               (r)ecv <type>\n");
                fprintf(stderr, "               (d)elete\n");
                fprintf(stderr, "               (m)ode <octal mode>\n");
                exit(1);
            }
```