University of Massachusetts Dartmouth
CIS 370, Fall 2013
10/15/2013, 10/17/2013
Lab 5: Inter-process Communication using Message Queues
Due: 10/22/2013 (Tuesday), 10/24/2013 (Thursday)

A. OBJECTIVE
Understand and experiment with Linux inter-process communication using message queues.

B. BASIC CONCEPTS
1) In Linux a Message Queue is implemented 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 is uniquely identified by an IPC identifier (an integer number), assigned by the kernel. The **`ipcs`** command can be used to obtain the status of all Linux IPC objects (Semaphores, Message Queues, Shared Memories).

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

By default, all three categories of objects are shown. Consider the following sample output of `ipcs`:

```
-bash-4.1 $ ipcs

        ------ Shared Memory Segments --------
        key        shmid      owner      perms      bytes      nattch     status
        0x00000000 1736704    gxie       600        393216     2          dest
        0x01eddf1b 1605633    gxie       600        1          0
        0x06e409c1 1638402    gxie       600        1          0
        0x0db58729 1671171    gxie       600        1          0
        0x3666b565 1703940    gxie       600        1          0
        0x00000000 1769477    gxie       600        393216     2          dest
        0x00000000 1802246    gxie       600        393216     2          dest
        0x00000000 1835015    gxie       600        393216     2          dest
        0x00000000 1867784    gxie       600        393216     2          dest
        0x00000000 1900553    gxie       600        393216     2          dest
        0x00000000 1933324    gxie       600        393216     2          dest
        0x00000000 1966093    gxie       600        393216     2          dest
        0x00000000 1998862    gxie       600        393216     2          dest
        0x00000000 2031631    gxie       600        393216     2          dest
        0x00000000 2064400    gxie       600        393216     2          dest
        0x00000000 2097169    gxie       600        393216     2          dest
        0x00000000 2129938    gxie       600        393216     2          dest
        0x00000000 2162707    gxie       600        393216     2          dest
        0x00000000 2195476    gxie       600        393216     2          dest
        0x00000000 2261013    gxie       600        393216     2          dest

        ------ Semaphore Arrays --------
        key        semid      owner      perms      nsems
        0x4143debb 98305      gxie       600        1
        0x0195da4b 131074     gxie       600        1

        ------ Message Queues --------
        key        msqid      owner      perms      used-bytes   messages
```

2) To obtain a unique identifier, a key must be used. The key must be mutually agreed upon by all processes that access the message queue. The key can be the same value every time, by hard-coding a key value into an application. This has the disadvantage of the key possibly being in use already. Often, the ftok() function is used to generate key values.

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 returned key value from ftok() is generated by combining the inode number and minor device number from the file in argument one, with the one character project identifier in the second argument.

```
key_t mykey;
mykey = ftok("/tmp/cis370", 't');
```

In this example, the directory `/tmp/cis370` is combined with the one letter identifier of `'t'` to generate a key.

3) To use a message queue, we must define a structure for the messages. It's totally up to the programmer to decide what to put into the structure, but the first field of the structure must be a `long` number that indicates the type of the message. For instance:

```
/* message buffer for msgsnd and msgrcv calls */
struct mymsgbuf
{
    long    msg_type;     /* type of message */
    long    sender_id;    /* sender identifier */
    char    msg[128];     /* content of the message */
};
```

The above structure shows that the message includes two fields: the first field is a sender ID and the second is a string for the message content. Basically, this means that any data can be sent via a message queue. However, you should keep in mind that there exists an internal limit of the maximum size of a given message. In Linux, this is defined in **`linux/msg.h`** as follows:

```
#define MSGMAX  4056   /* max size of message (bytes) */
```

Messages can be no larger than 4,056 bytes in total size, including the `msg_type` member, which is 4 bytes in length (long).

4) In order to create a new message queue, or access an existing queue, you use the **`msgget()`** system call.

SYSTEM CALL: msgget();

PROTOTYPE: int msgget (key_t key, int msgflg);
RETURNS: message queue identifier on success
-1 on error: errno = EACCESS (permission denied)
EEXIST (Queue exists, cannot create)
EIDRM (Queue is marked for deletion)
ENOENT (Queue does not exist)
ENOMEM (Not enough memory to create queue)
ENOSPC (Maximum queue limit exceeded)

The first argument to `msgget()` is the key value (returned by a call to `ftok()`). This key value is then compared to existing key values that exist within the kernel for other message queues. At that point, the open or access operation is dependent upon the contents of the `msgflg` argument.

`IPC_CREAT`: Create the queue if it doesn't already exist in the kernel.

`IPC_EXCL`: When used with `IPC_CREAT`, fail if queue already exists.

If `IPC_CREAT` is used alone, `msgget()` either returns the message queue identifier for a newly created message queue, or returns the identifier for a queue which exists with the same key value. If `IPC_EXCL` is used along with `IPC_CREAT`, then either a new queue is created, or if the queue 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 queue is opened for access. An optional octal mode may be OR'd into the mask, since each IPC object has permissions that are similar in functionality to file permissions on a UNIX file system! Look at the following function:

```
int open_queue( key_t keyval )
{
     int qid;

     if((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
     {
              return(-1);
     }

     return(qid);
}
```

It gets (or creates) a message queue with the given key (`keyval`) and the permission 0660. This function either returns a message queue identifier (`int`), or -1 on error.

5) Once we have the queue identifier, we can send or receive messages to or from the queue. To deliver a message to a queue, you use the `msgsnd()` system call:

SYSTEM CALL: msgsnd();

PROTOTYPE: int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg);
RETURNS: 0 on success
-1 on error: errno = EAGAIN (queue is full, and IPC_NOWAIT was asserted)
EACCES (permission denied, no write permission)
EFAULT (msgp address isn't accessible - invalid)
EIDRM (The message queue has been removed)
EINTR (Received a signal while waiting to write)
EINVAL (Invalid message queue identifier, non-positive message type, or invalid msg. size)
ENOMEM (Not enough memory to copy message buffer)

The first argument to `msgsnd()` is the queue identifier, returned by a previous call to `msgget`(). The second argument, `msgp`, is a pointer to our message buffer. The `msgsz` argument contains the size of the message in bytes, excluding the length of the message type (4 bytes long). The `msgflg` argument can be set to 0 (ignored), or:

`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.

```
int send_message( int qid, struct mymsgbuf *qbuf )
{
      int  result, length;

      /* The length is essentially the size of the structure - sizeof(mtype) */
      length = sizeof(struct mymsgbuf) - sizeof(long);

      /* Notice the type cast here!*/
      if((result = msgsnd( qid, (struct msgbuf *)qbuf, length, 0)) == -1)
           return(-1);

      return(result);
}
```

This small function attempts to send the message residing at the passed address (`qbuf`) to the message queue designated by the passed queue identifier (`qid`).

6) After creating/opening our message queue, we can retrieve a message from the queue by using the `msgrcv()` system call:

SYSTEM CALL: msgrcv();
PROTOTYPE: int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg);
RETURNS: Number of bytes copied into message buffer
-1 on error: errno = E2BIG (Message length is greater than msgsz, no MSG_NOERROR)
EACCES (No read permission)
EFAULT (Address pointed to by msgp is invalid)
EIDRM (Queue was removed during retrieval)
EINTR (Interrupted by arriving signal)
EINVAL (msgqid invalid, or msgsz less than 0)
ENOMSG (IPC_NOWAIT asserted, and no message exists in the queue to satisfy the request)

Obviously, 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. Once again, this can easily be calculated as:

```
msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

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.

If `IPC_NOWAIT` is passed as a flag, and no messages are available, the call returns `ENOMSG` to the calling process. Otherwise, the calling process blocks until a message arrives in the queue that satisfies the `msgrcv()` parameters. If the queue is deleted while a client is waiting on a message, `EIDRM` is returned. `EINTR` is returned if a signal is caught while the process is in the middle of blocking, and waiting for a message to arrive.

Let's look at another function for retrieving a message from our queue:

```
int read_message( int qid, long type, struct mymsgbuf *qbuf )
{
     int result, length;

     /* The length is essentially the size of the structure - sizeof(mtype) */
     length = sizeof(struct mymsgbuf) - sizeof(long);

     if((result = msgrcv( qid, (struct msgbuf *)qbuf, length, type,  0)) == -1)
     {
        return(-1);
     }

     return(result);
}
```

After successfully retrieving a message from the queue, the message entry within the queue is destroyed.

7) To perform control operations on a message queue, you use the `msgctl()` system call to first get a kernel structure that represents the queue.

SYSTEM CALL: msgctl();
PROTOTYPE: int msgctl (int msgqid, int cmd, struct msqid_ds *buf);
RETURNS: 0 on success
-1 on error: errno = EACCES (No read permission and cmd is IPC_STAT)
EFAULT (Address pointed to by buf is invalid with IPC_SET and IPC_STAT commands)
EIDRM (Queue was removed during retrieval)
EINVAL (msgqid invalid, or msgsz less than 0)
EPERM (IPC_SET or IPC_RMID command was issued, but calling process does not have write (alter) access to the queue)

The `msqid_ds` structure is defined as follows:

```
struct msqid_ds
{
    struct ipc_perm msg_perm; /* queue permissions */
    struct msg *msg_first;    /* first message on queue */
    struct msg *msg_last;     /* last message in queue */
    time_t msg_stime;         /* last msgsnd time */
    time_t msg_rtime;         /* last msgrcv time */
    time_t msg_ctime;         /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes;        /* max number of bytes on queue */
    ushort msg_lspid;         /* pid of last msgsnd */
    ushort msg_lrpid;         /* last receive pid */
};
```

You can use the following commands for the value of `cmd`:

`IPC_STAT`: Retrieves the `msqid_ds` structure for a queue, and stores it in the address of the `buf` argument.

`IPC_SET`: Sets the value of the `ipc_perm` member of the `msqid_ds` structure for a queue. Takes the values from the `buf` argument.

`IPC_RMID`: Removes the queue from the kernel.

The kernel maintains an instance of this `msqid_ds` structure for each queue in the system. By using the `IPC_STAT` command, we can retrieve a copy of this structure for examination:

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
    if( msgctl( qid, IPC_STAT, qbuf) == -1)
    {
        return(-1);
    }

    return(0);
}
```

If we are unable to copy the internal buffer, -1 will be returned to the calling function. If all go well, a value of 0 (zero) will be returned, and the passed buffer should contain a copy of the internal data structure for the message queue represented by the passed queue identifier (`qid`).

With a copy of the internal data structure for a queue, it seems that we can alter any attributes in this structure. Unfortunately, the only modifiable item in the data structure is the `ipc_perm` member. This contains the permissions for the queue, as well as information about the owner and creator. However, the only members of the `ipc_perm` structure that are modifiable are mode, uid, and gid. You can change the owner's user id, the owner's group id, and the access permissions for the queue. Look at the following function that changes the mode of a queue (The mode must be passed in as a character array (i.e. ``660")):

```
int change_queue_mode( int qid, char *mode )
{
     struct msqid_ds tmpbuf;

     /* Retrieve a current copy of the internal data structure */
     get_queue_ds( qid, &tmpbuf);

     /* Change the permissions using an old trick */
     sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);

     /* Update the internal data structure */
     if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
     {
         return(-1);
     }

     return(0);
}
```

We retrieve a current copy of the internal data structure by a quick call to the `get_queue_ds()` function. We then make a call to `sscanf()` to alter the mode member of the associated `msg_perm` structure. No changes take place, however, until the new copy is used to update the internal version. This duty is performed by a call to `msgctl()` using the `IPC_SET` command.

BE CAREFUL! It is possible to alter the permissions on a queue, and in doing so, inadvertently lock yourself out! Remember, these IPC objects don't go away unless they are properly removed, or the system is rebooted. So, even if you can't see a queue with `ipcs` doesn't mean that it isn't there.

After successfully retrieving a message from a queue, the message is removed. However, the message queue itself still exists within the kernel, available for later use, unless explicitly removed, or the system is rebooted. To completely remove a message queue from the kernel, you should call `msgctl()`, using the `IPC_RMID` command:

```
int remove_queue( int qid )
{
     if( msgctl( qid, IPC_RMID, 0) == -1)
     {
             return(-1);
     }

     return(0);
}
```

This function returns 0 if the queue was removed without incident, or else a value of -1. The removal of the queue is atomic in nature, and any subsequent accesses to the queue for whatever purpose will fail miserably.

C. REQUIREMENTS

1) In this lab, you are required to implement a program (using the above functions and the template code) called `lastname_msgqueue.c` for manipulating message queues and experiment with it. Your `lastname_msgqueue.c` program shall behave as follows.

- To send a message "text" with message type 'type':
  ```
  msgqueue s type "text"
  ```
- To receive and display a message of type 'type':
  ```
  msgqueue r type
  ```
- To change the permission mode to 'mode' (an octal number):
  ```
  msgqueue m mode
  ```
- To delete the message queue:
  ```
  msgqueue d
  ```
- If no commands are provided, your program shall print the following:
  ```
  USAGE:  msgqueue   (s)end  <type> <message>
                     (r)ecv  <type>
                     (d)elete
                     (m)ode  <mode>
  ```

2) Download the template code: http://www.cis.umassd.edu/~jplante/cis370/lab05/msgqueue.c