Basic Queues

Mark Sitkowski C.Eng, M.I.E.E
http://www.designsim.com.au

Introduction

It occasionally happens, that our incoming, or outgoing data cannot be processed as it is generated or, for some reason, we choose to process it at a later time.
A typical example might be a client-server system, where it is necessary to queue the socket descriptors of incoming connections, because of some limit on the number of active processes, or a message hub, which accepts data synchronously, but must rely on other processes to remove the data asynchronously.
Apart from the numerous commercially-available third party implementations of queuing systems, Unix has two highly efficient queuing mechanisms, which can be used for extremely low overhead systems of queues.

Kernel mode queues

The kernel uses queues internally, for the implementation of functions such as device drivers, and the system call interface to this mechanism is available for the implementation of application programs.
The queues so produced are implemented in memory, so they are very fast but, because there is no permanent storage of the data, these queues are also non-persistent. If the process, or the machine crashes, all of the queued data will be lost, and all incoming data will never be enqueued.
A queuing system, based on kernel queues, is the subject of 'Advanced Queues', while here, we will concentrate on disk-based user mode queues.

User mode queues

The kernel mode queuing system is a little Spartan, and it is sometimes more convenient to use the user mode queue library functions, which offer a little more functionality, namely:

- Notification of message arrival, by sending a signal to the monitoring process.
- Prioritisation of messages

There are only four fundamental commands to remember:

mq_open() - opens an existing queue, or creates a new queue
mq_send() - enqueues a message
mq_receive() - dequeues a message
mq_notify() - notifies a process of the arrival of a message

The remaining five commands perform housekeeping tasks:

mq_close() - closes a queue
mq_unlink() - deletes a queue from the disk
mq_getattr() - interrogates a queue's characteristics
mq_setattr() - sets a queue's characteristics

A single structure definition is used to set and get the queues' attributes, and is defined as:

```
struct mq_attr {
   long    mq_flags        /* message queue flags */
   long    mq_maxmsg       /* maximum number of messages */
   long    mq_msgsize      /* maximum message size */
   long    mq_curmsgs      /*  number of messages currently queued */
};
```

The mq series of commands all relate to disk based queues. The queues themselves, are created in the /tmp directory, and are always referred to in the commands, as if they were situated below the root directory.
Thus, to create a queue called 'zq', we would call mq_open(), like this:

```
Int qd;
struct mq_attr atr;

   atr.mq_maxmsg = 100;
   atr.mq_msgsize = 255;

   if((qd = mq_open("/zq", O_RDWR|O_CREAT, 0755, &atr)) == (mqd_t)-1){
      perror("mq_open");
   }
```

Notice the similarity between the above syntax, and that of the 'open()' command, for a file. The returned value is the queue descriptor, while the flags are exactly the same, as defined in fcntl.h for those relating to a file. The pointer to the 'atr' structure permits the setting of the maximum number of messages, and the maximum message size, prior to calling mq_open.

Enqueuing a message is analogous to a 'write()' on a file:

```
char *msg = "xyz";
int priority = 5;

   if(mq_send(qd, msg, strlen(msg), priority) == -1){
      perror("mq_send");
   }
```

The extra parameter, 'priority' determines the order, in which the message will be removed from the queue, when it is dequeued, with '1' being the highest priority.

The dequeuing is performed by mq_receive():

```
unsigned char data[8192];
int priority;
int n;

   If((n = mq_receive(qd, (char *)data, sizeof(data), &priority)) > 0){
      Printf("Received %d byte message >%s< with %d priority\n", n, data, priority);
    }
```

Messages are taken off the queue, in order of their priority, which is returned by mq_receive(), into the variable passed to it. The return value of the function is the number of bytes in the message.
In normal operation, this function would be called in a 'while' loop, and the queue length would be checked at each iteration of the loop. The checking is done with the mq_getattr() function, called with the queue descriptor, and the atr structure, defined above:

```
if(mq_getattr(qd, &atr)  == 0){
   if(atr.mq_curmsgs == 0){
      printf("No more messages\n");
       mq_close(qd);
   }
}
```

The following code extract puts this all together:

```
while((rval = mq_receive(qd, (char *)data, sizeof(data), &priority)) > 0){
   printf("Client received: >%s< priority %d\n", data, priority);
   memset(data, '\0', sizeof(data));
   if(mq_getattr(qd, &atr)  == 0){
      if(atr.mq_curmsgs == 0){
         printf("No more messages\n");
         mq_close(qd);
         break;
      }
   }
}
```

We now have all the information we need, to write a little test program, which exercises all of these queuing functions. Instead of attempting to re-create MQ Series from scratch (which we will leave for the 'Advanced Queues' article), this program merely does the following:

1. Create a queue, whose descriptor is 'qd'.
2. Launch a child process, 'child()' which asks to be notified of the arrival of a message
3. Enqueue 4 messages, in ascending order of priority.
4. The child pulls the messages off the queue, in the order that they arrived, i.e, in order of priority. It then quits.
5. Launch another child process, 'client()', which merely performs a blocking read of the queue.
6. Enqueue 4 more messages, in descending order of priority
7. The child, again, pulls the messages off, in order of priority, which means the reverse of the order of their arrival. It does not quit.

The notification mechanism is by means of a software interrupt, defined by means of the sigevent structure. We first create the variable:

```
struct sigevent ev;
```

The interesting parts of this structure (defined fully in siginfo.h) are

```
struct sigevent {
    int sigev_notify;
```

```
    int  sigev_signo;
}
```

where sigev_notify has the values

SIGEV_NONE
SIGEV_SIGNAL
SIGEV_THREAD

We will choose SIGEV_SIGNAL, since we want to catch an interrupt, with the arrival of each message on our queue. Later, if we need to turn off notification, we can do it by passing in SIGEV_NONE.
Since sigev_signo lets us choose which signal can be sent to us, we'll choose something safe, that isn't used by other processes. SIGURG is normally sent out when an urgent condition exists on a socket or other I/O device and, in that capacity, is of no interest to us. Therefore, we will use SIGURG, and register it, together with our interrupt handler, in main():

```
   signal(SIGURG, interrupt);
```

Then, in our child() function, when our child process is running, we define the kind of event we need, and the signal number that we're expecting, as follows:

```
        ev.sigev_notify = SIGEV_SIGNAL;
        ev.sigev_signo = SIGURG;
```

Immediately after these lines, we call pause(), which puts the process into a catatonic state, waiting for the arrival of an interrupt.

In reality, the server and client code would probably be in separate files, and run in unrelated processes. Since this is merely an exercise, all of the code is in one file, as follows.

Server and Client Code

```
#include <mqueue.h>
#include <sys/stream.h>
#include <sys/ddi.h>

void interrupt(int);       /* interrupt handler */

struct mq_attr atr;

char *msg1 = "Mary had a little lamb\n";
char *msg2 = "She also had a duck\n";
char *msg3 = "She put them on the mantelpiece\n";
char *msg4 = "To see if they would fall\n";

char *msg5 = "Mary had a little lamb\n";
char *msg6 = "full of fun and frolicks\n";
char *msg7 = "She threw it up into the air\n";
char *msg8 = "And caught it by its tail\n";

mqd_t qd;
```

```
main()                      /* main */

{

char data[255];
unsigned int priority;
int rval;
pid_t pid;
struct mq_attr xatr;

    signal(SIGURG, interrupt);

    atr.mq_maxmsg = 100;
    atr.mq_msgsize = 255;

    if((qd = mq_open("/zq", O_RDWR|O_CREAT, 0755, &atr)) == (mqd_t)-
1){
        perror("mq_open");
    }
    pid = child(qd);       /* this asks to get notified */

    sleep(1);        /* give the child time to stabilise */

    /* queue ordering is by priority, not time of arrival */
    if(mq_send(qd, msg4, strlen(msg4), 5) == -1){
        perror("mq_send");
    }
    if(mq_send(qd, msg3, strlen(msg3), 6) == -1){
        perror("mq_send");
    }
    if(mq_send(qd, msg2, strlen(msg2), 7) == -1){
        perror("mq_send");
    }
    if(mq_send(qd, msg1, strlen(msg1), 8) == -1){
        perror("mq_send");
    }

    sleep(1);        /* give the child time to exit */
    pid = client();  /* blocking, but no notification */

    /* these must arrive after the queue empties, or the child won't
exit */
    if(mq_send(qd, msg5, strlen(msg5), 4) == -1){
        perror("mq_send");
    }
    if(mq_send(qd, msg6, strlen(msg6), 3) == -1){
        perror("mq_send");
    }
    if(mq_send(qd, msg7, strlen(msg7), 2) == -1){
        perror("mq_send");
    }
    if(mq_send(qd, msg8, strlen(msg8), 1) == -1){
        perror("mq_send");
    }

}                           /* main */

/*****************************************************************
•  Simple blocking read loop, which checks the queue length at each
•   pass, and exits when it's empty.
 *****************************************************************/
```

```
client()                    /* client */

{

pid_t pid;
char data[255];
unsigned int priority;
int rval;

    switch((pid = fork())){
        case -1:
        break;
        case 0:
            printf("Client collecting messages from queue...\n");
            /* this will block until the first msg arrives */
            while((rval = mq_receive(qd, (char *)data, sizeof(data),
&priority)) > 0){
                printf("Client received: >%s< priority %d\n", data,
priority);
                memset(data, '\0', sizeof(data));
                if(mq_getattr(qd, &atr)  == 0){
                    if(atr.mq_curmsgs == 0){
                        printf("No more messages\n");
                        mq_close(qd);
                        break;
                    }
                } else {
                    perror("mq_getattr");
                    break;
                }
            }
            printf("Done\n");
            mq_unlink("/zq");
            exit(0);
        break;
        default:
            return(pid);
        break;
    }

}                          /* client */

/*****************************************************************
•  The child asks to be notified of the arrival of a message, by
•  means of SIGURG, for which we've defined a handler. The child
•  then calls pause(), and waits for an interrupt. Inside the
•  interrupt handler, it performs blocking reads on the queue,
•  checking its length each time. When the queue is empty, it
•  returns, and calls mq_notify again, to turn off notification,
*  and permit the client routine to access the queue.
*****************************************************************/

child(qqd)                     /* child */

mqd_t qqd;

{

struct sigevent ev;
```

```
pid_t pid;

    switch((pid = fork())){
        case -1:
        break;
        case 0:
            printf("Child collecting messages from queue...\n");
            ev.sigev_notify = SIGEV_SIGNAL;
            ev.sigev_signo = SIGURG;

            if(mq_notify(qqd, &ev) < 0){
                perror("mq_notify");
            }
            pause();
            if(mq_notify(qqd, NULL) < 0){
                perror("mq_notify");
            }
            exit(0);
        break;
        default:
            return(pid);
        break;
    }


}                       /* child */

/*****************************************************************
•  Interrupt handler
•  We're only interested in SIGURG, for which we've been waiting
•  in pause().We perform our dequeuing function in this handler,
•  to save ourselves a function call, so It is important that the
•   queue variables be visible globally.
•  The mq_receive() loop performs reads the queue, checking its
•  Length each time. When the queue is empty, we return.
 *****************************************************************/

void
interrupt(what)           /* interrupt */

int what;

{

char data[255];
unsigned int priority;
int rval;

    printf("Received signal %d...\n", what);

    switch(what){
        case SIGURG:
            while((rval = mq_receive(qd, (char *)data, sizeof(data),
&priority)) > 0){
                printf("Child received: >%s< priority %d\n", data,
priority);
                memset(data, '\0', sizeof(data));
                if(mq_getattr(qd, &atr)  == 0){
                    if(atr.mq_curmsgs == 0){
                        printf("No more messages\n");
```

```
                        break;
                    }
                } else {
                    perror("mq_getattr");
                    break;
                }
            }
        break;
    }

}                           /* interrupt */
```