Message Queues

Those people who brought us System V have seen fit to include some IPC goodies that have been implemented on various platforms (including Linux, of course.) This document describes the usage and functionality of the extremely groovy System V Message Queues!

As usual, I want to spew some overview at you before getting into the nitty-gritty. A message queue works kind of like a FIFO, but supports some additional functionality. Generally, see, messages are taken off the queue in the order they are put on. Specifically, however, there are ways to pull certain messages out of the queue before they reach the front. It's like cutting in line. (Incidentally, don't try to cut in line while visiting the Great America amusement park in Silicon Valley, as you can be arrested for it. They take cutting *very* seriously down there.)

In terms of usage, a process can create a new message queue, or it can connect to an existing one. In this, the latter, way two processes can exchange information through the same message queue. Score.

One more thing about System V IPC: when you create a message queue, it doesn't go away until you destroy it. All the processes that have ever used it can quit, but the queue will still exist. A good practice is to use the `ipcs` command to check if any of your unused message queues are just floating around out there. You can destroy them with the `ipcrm` command, which is preferable to getting a visit from the sysadmin telling you that you've grabbed every available message queue on the system.

Where's my queue?

Let's get something going! First of all, you want to connect to a queue, or create it if it doesn't exist. The call to accomplish this is the `msgget()` system call:

```
int msgget(key_t key, int msgflg);
```

`msgget()` returns the message queue ID on success, or `-1` on failure (and it sets `errno`, of course.)

The arguments are a little weird, but can be understood with a little brow-beating. The first, *`key`* is a system-wide unique identifier describing the queue you want to connect to (or create). Every other process that wants to connect to this queue will have to use the same *`key`*.

The other argument, *`msgflg`* tells `msgget()` what to do with queue in question. To create a queue, this field must be set equal to `IPC_CREAT` bit-wise OR'd with the permissions for this queue. (The queue permissions are the same as standard file permissions--queues take on the user-id and group-id of the program that created them.)

A sample call is given in the following section.

"Are you the Key Master?"

What about this *key* nonsense? How do we create one? Well, since the type `key_t` is actually just a `long`, you can use any number you want. But what if you hard-code the number and some other unrelated program hardcodes the same number but wants another queue? The solution is to use the `ftok()` function which generates a key from two arguments:

```
key_t ftok(const char *path, int id);
```

Ok, this is getting weird. Basically, *path* just has to be a file that this process can read. The other argument, *id* is usually just set to some arbitrary char, like 'A'. The `ftok()` function uses information about the named file (like inode number, etc.) and the *id* to generate a probably-unique *key* for `msgget()`. Programs that want to use the same queue must generate the same *key*, so they must pass the same parameters to `ftok()`.

Finally, it's time to make the call:

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

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
```

In the above example, I set the permissions on the queue to `666` (or `rw-rw-rw-`, if that makes more sense to you). And now we have `msqid` which will be used to send and receive messages from the queue.

Sending to the queue

Once you've connected to the message queue using `msgget()`, you are ready to send and receive messages. First, the sending:

Each message is made up of two parts, which are defined in the template structure `struct msgbuf`, as defined in `sys/msg.h`:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

The field `mtype` is used later when retrieving messages from the queue, and can be set to any positive number. `mtext` is the data this will be added to the queue.

"What?! You can only put one byte arrays onto a message queue?! Worthless!!" Well, not exactly. You can use any structure you want to put messages on the queue, as long as the first element is a long. For instance, we could use this structure to store all kinds of goodies:

```
struct pirate_msgbuf {
    long mtype;  /* must be positive */
    char name[30];
    char ship_type;
    int notoriety;
    int cruelty;
```

```
        int booty_value;
    };
```

Ok, so how do we pass this information to a message queue? The answer is simple, my friends: just use `msgsnd()`:

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

`msqid` is the message queue identifier returned by `msgget()`. The pointer *`msgp`* is a pointer to the data you want to put on the queue. *`msgsz`* is the size in bytes of the data to add to the queue. Finally, *`msgflg`* allows you to set some optional flag parameters, which we'll ignore for now by setting it to `0`.

And here is a code snippet that shows one of our pirate structures being added to the message queue:

```
#include

key_t key;
int msqid;
struct pirate_msgbuf pmb = {2, "L'Olonais", 'S', 80, 10, 12035};

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

msgsnd(msqid, &pmb, sizeof(pmb), 0);  /* stick him on the queue */
```

Aside from remembering to error-check the return values from all these functions, this is all there is to it. Oh, yeah: note that I arbitrarily set the `mtype` field to `2` up there. That'll be important in the next section.

Receiving from the queue

Now that we have the dreaded pirate [Francis L'Olonais] stuck in our message queue, how do we get him out? As you can imagine, there is a counterpart to `msgsnd()`: it is `msgrcv()`. How imaginative.

A call to `msgrcv()` that would do it looks something like this:

```
#include

key_t key;
int msqid;
struct pirate_msgbuf pmb; /* where L'Olonais is to be kept */

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

msgrcv(msqid, &pmb, sizeof(pmb), 2, 0);  /* get him off the queue! */
```

There is something new to note in the `msgrcv()` call: the `2`! What does it mean? Here's the synopsis of the call:

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

The `2` we specified in the call is the requested *msgtyp*. Recall that we set the `mtype` arbitrarily to `2` in the `msgsnd()` section of this document, so that will be the one that is retrieved from the queue.

Actually, the behavior of `msgrcv()` can be modified drastically by choosing a *msgtyp* that is positive, negative, or zero:

msgtyp	**Effect on `msgrcv()`**
Zero	Retrieve the next message on the queue, regardless of its `mtype`.
Positive	Get the next message with an `mtype` **equal to** the specified *msgtyp*.
Negative	Retrieve the first message on the queue whose `mtype` field is less than or equal to the absolute value of the *msgtyp* argument.

Table 1. Effects of the *msgtyp* argument on `msgrcv()`.

So, what will often be the case is that you'll simply want the next message on the queue, no matter what `mtype` it is. As such, you'd set the *msgtyp* parameter to `0`.

Destroying a message queue

There comes a time when you have to destroy a message queue. Like I said before, they will stick around until you explicitly remove them; it is important that you do this so you don't waste system resources. Ok, so you've been using this message queue all day, and it's getting old. You want to obliterate it. There are two ways:

1. Use the Unix command `ipcs` to get a list of defined message queues, then use the command `ipcrm` to delete the queue.
2. Write a program to do it for you.

Often, the latter choice is the most appropriate, since you might want your program to clean up the queue at some time or another. To do this requires the introduction of another function: `msgctl()`.

The synopsis of `msgctl()` is:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Of course, *msqid* is the queue identifier obtained from `msgget()`. The important argument is **`cmd`** which tells `msgctl()` how to behave. It can be a variety of things, but we're only going to talk about `IPC_RMID`, which is used to remove the message queue. The *buf* argument can be set to `NULL` for the purposes of `IPC_RMID`.

Say that we have the queue we created above to hold the pirates. You can destroy that queue by issuing the following call:

```
#include
.
.
msgctl(msqid, IPC_RMID, NULL);
```

And the message queue is no more.

Sample programs, anyone?

For the sake of completeness, I'll include a brace of programs that will communicate using message queues. The first, `kirk.c` adds messages to the message queue, and `spock.c` retrieves them.

Here is the source for kirk.c:

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

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }

    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1; /* we don't really care in this case */
```

```
    while(gets(buf.mtext), !feof(stdin)) {
        if (msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0) == -1)
            perror("msgsnd");
    }

    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }

    return 0;
}
```

The way `kirk` works is that it allows you to enter lines of text. Each line is bundled into a message and added to the message queue. The message queue is then read by `spock`.

Here is the source for spock.c:

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

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {  /* same key as kirk.c */
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */
        perror("msgget");
        exit(1);
    }

    printf("spock: ready to receive messages, captain.\n");
```

```
        for(;;) { /* Spock never quits! */
            if (msgrcv(msqid, (struct msgbuf *)&buf, sizeof(buf), 0, 0) == -1) {
                perror("msgrcv");
                exit(1);
            }
            printf("spock: \"%s\"\n", buf.mtext);
        }

        return 0;
    }
```

Notice that `spock`, in the call to `msgget()`, doesn't include the `IPC_CREAT` option. We've left it up to `kirk` to create the message queue, and `spock` will return an error if he hasn't done so.

Notice what happens when you're running both in separate windows and you kill one or the other. Also try running two copies of `kirk` or two copies of `spock` to get an idea of what happens when you have two readers or two writers. Another interesting demonstration is to run `kirk`, enter a bunch of messages, then run `spock` and see it retrieve all the messages in one swoop. Just messing around with these toy programs will help you gain an understanding of what is really going on.

Conclusions

There is more to message queues than this short tutorial can present. Be sure to look in the man pages to see what else you can do, especially in the area of `msgctl()`. Also, there are more options you can pass to other functions to control how `msgsnd()` and `msgrcv()` handle if the queue is full or empty, respectively.

HPUX man pages

If you don't run HPUX, be sure to check your local man pages!

- `ftok()`
- `ipcs`
- `ipcrm`
- `msgctl()`
- `msgget()`
- `msgsnd()`
