Operating Systems

Lab 02： Message Queue, IPC

Goals:

Learn to work with message queue to implement inter-process communication.

Policies:

It should go without saying that all the work that you will turn in for this lab will be yours. Do not surf the web to get inspiration for this assignment, do not include code that was not written by you. You should try your best to debug your code on your own, but it's fine to get help from a colleague as long as that means getting assistance to identify the problem and doesn't go as far as receiving source code to fix it (in writing or orally).

Contents:

1. What is message queue?

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.

2. Creating A Message Queue - msgget()

A new queue is created or an existing queue opened by **msgget()**.

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

msgget() returns the message queue ID on success, or -1 on failure (upon failure, the external variable *errno* is set to indicate the reason for failure).

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.

The value passed to the msgflg argument must be an integer-type value that will specify the following:

- Operations Permissions
- Control Fields (commands)

Operation permissions determine the operations that processes are permitted to perform on the associated message queue. "Read" permission is necessary for receiving messages or for determining queue status by means of a msgctl IPC_STAT operation. "Write" permission is necessary for sending messages.

The following table reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation permissions codes

Operation permissions	Octal value
Read by user	00400
Write by user	00200
Read by group	00040
Write by group	00020
Read by others	00004
Write by others	00002

A specific value is derived by adding or bitwise ORing the octal values for the operation permissions wanted. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006).

Control Fields (commands) are predefined constants (represented by all upper-case letters). The flags which apply to the msgget system call are IPC_CREAT and IPC_EXCL.

The value for **msgflg** is therefore a combination of **operation permissions** and **control commands**. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by adding or bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

For example, to create a queue, the control field must be set equal to IPC_CREAT bitwise OR'd with the operation permissions for this queue.

For example:

```
msqid = msgget(key, 00660 | IPC_CREAT);
```

Here 0666 represents:

- 0: this is an octal number
- 6: 00600 = 00400 + 00200. 00600 represents current user can read and write.
- 6: 00060 = 00040 + 00020. 00060 represents can be read and written by a group.
- 0: 00000 = 00000 + 00000. 00000 represents other users cannot read and write.

3. **ftok(): used to generate a unique key.**

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);
```

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().

An example for using ftok():

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

#define IPCKEY 0x11

int main( void )
{
    int i=0;
    for ( i = 1; i < 256; ++ i )
        printf( "i:%d, key = %x\n", i, ftok( "/tmp", i ) );

    return 0;
}
```

Execution results:

```
zaobohe@ubuntu:~/Desktop$ gcc ftok1.c -o ftok1
zaobohe@ubuntu:~/Desktop$ ./ftok1
i:1, key = 1050003
i:2, key = 2050003
i:3, key = 3050003
i:4, key = 4050003
i:5, key = 5050003
i:6, key = 6050003
i:7, key = 7050003
i:8, key = 8050003
i:9, key = 9050003
i:10, key = a050003
i:11, key = b050003
i:12, key = c050003
i:13, key = d050003
i:14, key = e050003
i:15, key = f050003
i:16, key = 10050003
i:17, key = 11050003
i:18, key = 12050003
i:19, key = 13050003
i:20, key = 14050003
i:21, key = 15050003
i:22, key = 16050003
i:23, key = 17050003
```

Finally, it's time to make the call:

```
// ftok to generate unique key
key = ftok("progfile", 65);

// msgget creates a message queue
// and returns identifier
msgid = msgget(key, 0666 | IPC_CREAT);
```

4. 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[100];
};
```

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.

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 (not counting the size of the mtype member). Finally, msgflg allows you to set some optional flag parameters, which we'll ignore for now by setting it to 0.

For example:

```
// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

msgid = msgget(key, 0666 | IPC_CREAT);

// msgsnd to send message
msgsnd(msgid, &message, sizeof(message), 0);
```

5. Receiving from the queue

Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

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

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

key_t key;
int msqid;
struct pirate_msgbuf pmb;

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

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

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.

6. Destroying a message queue

There comes a time when you have to destroy a message queue, since 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.

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 data. You can destroy that queue by issuing the following call:

```
#include <sys/msg.h>
.
.
msgctl(msqid, IPC_RMID, NULL);
```

And the message queue is no more.

7. Examples:

```
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 10

// structure for message queue
struct mesg_buffer {
        long mesg_type;
        char mesg_text[100];
} message;

int main()
{
        key_t key;
        int msgid;

        // ftok to generate unique key
        key = ftok("progfile", 65);

        // msgget creates a message queue
        // and returns identifier
        msgid = msgget(key, 0666 | IPC_CREAT);
        message.mesg_type = 1;
```

```
        printf("Write Data : ");
        fgets(message.mesg_text,MAX,stdin);

        // msgsnd to send message
        msgsnd(msgid, &message, sizeof(message), 0);

        // display the message
        printf("Data send is : %s \n", message.mesg_text);

        return 0;
}
```

```
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n",
                    message.mesg_text);

    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

Execution results:

```
zaobohe@ubuntu:~/Desktop$ gcc mqw.c -o mqw
zaobohe@ubuntu:~/Desktop$ ./mqw
Write Data : zaobohe
Data send is : zaobohe

zaobohe@ubuntu:~/Desktop$ gcc mqr.c -o mqr
zaobohe@ubuntu:~/Desktop$ ./mqr
Data Received is : zaobohe

zaobohe@ubuntu:~/Desktop$ a
```

8. What should you do:

The writer (or sender) program has been given: **sender.c**.

Please write the reader (or receiver) program, named as: **receiver.c**

Like the given example, please print the received messages by the reader.

9. What should you submit:

1) Please submit a zip file which includes:

a) the C code of receiver.c:

b) A report to show: your idea, your code structure, and execution results.

2) Your zip file should be named as: "姓名_lab2"

3) Please submit your zip file by sending email to me as attachment: my email address is: hzb564@jnu.edu.cn