ITF22519: Introduction to Operating Systems

Fall Semester, 2022

Lab12: InterProcess Communication (IPC) 2

Submission Deadline: November 15th, 2022 23:59

In previous lab, you have learned how two different processes in the same computer use **Pipe, Signal** and **Shared Memory** to communicate. In this lab, we will look at how these processes use **Message Queue** to coordinate their activities. In Linux systems, there are two kinds of message queues: System V message queues and POSIX message queues but we only focus on POSIX message queues.

1 Message Queues

When processes are in the same system, they can communicate through a material in the common memory location where they all can access. Pipe, FIFO (named pipe) and Shared Memory are on this category. In addition, the processes can also communicate by using massage passing. **Message Queue** falls on this category.

Message queues are a way of passing priority messages from one process to another. Similar to previous lab, we need two processes. One process, called sending process, sends messages to the queue. The other process, called receiving process, retrieves the message from the queue. In POSIX message queues, all messages are sent and received in their entirety. Note that each message has its own priority. The priority is indicated by non-negative value with 0 being the lowest. When messages are put into the queue, the ones with highest priority will be at the front of the queue. To implement the sending process, we need a program which creates a message queue and writes a message (messages) to the message queue. The program for receiving process read messages from message queue and finally free the message queue. POSIX message queue APIs provide functions for those activities. Use man page `mq_overview(7)`, `mq_open(3)`, `mq_send(3)`, `mq_receive(3)` and `mq_notify(3)` for more information. One nice feature of message queues is the ability to subscribe to events on the queue and to handle the events asynchronously.

1.1 Message Queues Implementation

- **mq_open():**

First, a name for a message queue is needed so that other processes can refer to the queue. The name starts with a `/` followed by a number of characters. POSIX API provides `mq_open()` for this purpose. The function is executed in the sending process. The `mq_open()` creates a new POSIX message queue or opens an existing queue. The syntax of `mq_open()` is as followed:

```
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

The `oflag` argument specifies flags that control the operation of the call. Some examples of `oflag` are[Manpage];

- O_RDONLY: Open the queue to receive messages only.

- O_WRONLY: Open the queue to send messages only.
- O_RDWR: Open the queue to both send and receive messages.
- O_CREAT:Create the message queue if it does not exist
- O_EXCL: If O_CREAT was specified in `oflag`, and a queue with the given name already exists, then fail with the error EEXIST.

In order to use `mq_open()`, you need to add the following header files:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
```

Here is one example of using `mq_open()`

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
...
mq_open ("/Introduction2OS", O_CREAT | O_WRONLY, 0600, NULL);
...
```

- **mq_send():**

The sending process has to send a message to the queue. This is done by using the `mq_send()`. This function is defined in the header file:

```
#include <mqueue.h>
```

The syntax of `mq_send()` is as followed:

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);
```

- mqdes: descripter for the message queue.
- *msg_ptr is a pointer pointing to the message to be sent.
- msg_len: length of the message
- msg_prio: priority of the message which is an unsigned integer as mentioned above.

Here is one example of `mq_send()`:

```
mq_send (mqd, "HELLO", 6, 15);
```

This function sends the message "HELLO" with the size of 6 including null character and priority of 15.

- **mq_receive():**

The `mq_receive()` is executed by receiving process to retrieves a message from a queue. The syntax of `mq_send()` is as followed:

```
int mq_receive(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);
```

The arguments are the same as that of `mq_send()`. Example of `mq_receive()`

```
mq_receive (mqd, buffer, attr.mq_msgsize, &priority);
```

- **mq_close() and mq_unlink():**

`mq_close()` and `mq_unlink()` are executed in the receiving process. The `mq_close()` is to close the message queue. If you want to delete the message queue, you can use `mq_unlink`. The syntax of these two functions are as follows:

```
int mq_close(mqd_t mqdes);
int mq_unlink(const char *name);
```

These two functions are declared in the header file `mqueue.h`. The `mq_close()` closes the message queue descriptor `mqdes`. The `mq_unlink()` removes the specified message queue name. The message queue name is removed immediately. The queue itself is destroyed once any other processes that have the queue open close their descriptors referring to the queue[Manpage].

To compile a program using massge queue, you need to link with `-lrt` flag which is mentioned in the previous lab.

Task 1

Compile the files `mq_send.c` and `mq_receive.c` with the `-lrt` flag. Open two terminals. In the first termial, start `mq_send` and then in the other, start `mq_receive`.

- How do two programs work?
- Run another time and report what happens.

Task 2

Compile the files `mq_sendingPro.c` and `mq_receivingPro.c` with the `-lrt` flag to create two executable files named `mq_sending` and `mq_receiving`, repectively. Open two terminals. In the first termial, start `mq_sending` and then in the other start `mq_receiving`. In your report answer the following questions:

- What is the output from each program?
- What happens if you start them in the opposite order?
- Change `mq_receivingPro.c` to send a second message which is "My name is X" where "X" is your name. Change `mq_sendingPro.c` to wait for and print this second message before exiting.

2 Exercises

2.1 Exercise 1 (50 pts)

Write two programs `Ex1_send.c` and `Ex1_receive.c` will be run on two terminals:

- `Ex1_send.c` sends several messages each with a random priority to a message queue.
- `Ex1_receive.c` retrieves messages from the message queue.
- Run two programs in two terminals and copy screenshot in your output?
- What is the order of received messages?

2.2 Exercise 2 (50 pts)

Write two programs `read_mq.c` and `write_mq.c` will be run on two terminals:

- `write_mq.c` while true, reads one line that user enters into one terminal and send it to a message queue.
- `read_mq.c` while true reads the message from the same message queue and print it out to the other terminal.
- Do we need to synchronize the two programs to ensure read operations will always happen after write operations?

3 What To Submit

Upload related files and your report for this lab to your GitHub repository.