COP 5615: Distributed Operating Systems Principles

Fall 2016

Professor Sumi Helal

Lab 2: Extended Message Passing in Xinu

Due: 1:00pm EST Tuesday October 4, 2016

I. Objectives

- You will learn how processes communicate with each other by passing messages, and understand potential issues during communication.
- You will learn how to create new message passing system calls, by extending the Xinu messaging interface to allow two special cases: (a) multiple messages to be sent and received to/by a single process, and (b) a message to be sent and received to/by multiple processes.

II. Background - communicating by message passing

Processes can communicate with each other by sending and receiving messages. Sender process creates a message and sends it to a specific receiver process. The receiver receives the message and then waits until the next message comes from any senders. In order to implement message passing, the Xinu process table is used; a flag (process has message) is used in addition to storage in the table to store the message itself. For message passing, Xinu provides a send/receive set of system calls as follows:

- send - send a message to a process
- receive – receive a message from some other process
- recvclr - check if a message has been received from another process and return it if it exists, but do not wait if one does not exist. Recvclr removes (clears) the message from the process table if one exists.
- recvtime - same as receive, yet will only wait for a limited specified time for the message to be received.

III. Extending send/receive system calls

In this lab assignment, you will extend Xinu message passing and will implement your own system calls for handling two capabilities. For the first, your system call will allow sending multiple messages to a single process. In the second, a sender process will be able to send a message to multiple receivers.

A. System calls for sending/receiving multiple messages

A sender process may need to send multiple messages to a single process. For instance, different types of messages need to be sent at the same time. Or a message should be segmented in a fixed size and then sent to a receiver process. The messages should be received in the order in which they were sent. You may use a circular buffer for queuing (or dequeuing) the messages as you did in Lab 1. You will be implementing the following system calls:

syscall **sendMsg** (pid32 *pid*, umsg32 *msg*);	Sending message (*msg*) to process (*pid*). In case a message or a number of messages are already waiting on *pid* to receive them, the new *msg* will be queued. It will return OK on success or SYSERR on error.
umsg32 **receiveMsg** (*void*);	Receiving a message from any sender. It causes the calling process to wait for a message to be sent (by any process). When a message is sent, it is received and returned.
uint32 **sendMsgs** (pid32 *pid*, umsg32* *msgs*, uint32 *msg_count*);	Sending a group (*msg_count*) of messages (*msgs*) to process (*pid*). It will return the number of *msgs* successfully sent or SYSERR on error.
syscall **receiveMsgs** (umsg32* *msgs*, uint32 *msg_count*);	Receiving a group (*msg_count*) of messages (msgs). It causes the calling process to wait for msg_count messages to be sent. When all messages are in the queue, they are then all together immediately received.

B. System call for sending to multiple recipients

Sometimes a sender process is required to send a message to multiple processes. For instance, after a process changes data, it should update the change in other processes which use the data. You will be implementing the following system calls;

uint32 **sendnMsg** (uint32 *pid_count,* pid32* *pids*, umsg32 *msg*);	Sending the message (*msg*) to the number (*pid_count*) of processes (*pids*). It will return the number of *pids* successfully sent to or SYSERR on error.

C. Requirements and notes

1. You need to add your new system calls in Xinu by updating include/prototypes.h. Insert the following:

```
extern syscall sendMsg (pid32, umsg32);
extern unit32 sendMsgs (pid32, umsg32*, uint32);
extern unit32 sendnMsg (uint32, pid32*, umsg32,);
extern umsg32 receiveMsg (void);
extern syscall receiveMsgs (umsg32*, unit32);
```

2. Do not change the existing messaging system calls (send, receive, recvclr, and recvtime), because they are used by other system processes. Also do not change the process table entry in any way that causes existing system calls to no longer work. If changes are required, make sure to only add onto the existing structure; never remove any existing variables.

3. The maximum number of messages is 10. Also make the number of recipient processes no more than 3 for each of the senders and receivers (3 senders and 3 receivers).

4. Processes should print message(s) sent or received while the program is running. Senders should show recipients' pid and msgs, and how many messages successfully sent. In receivers' side, received messages should be displayed. Make sure you indicate in the print message whether a message is being sent or has been received.

5. You are able to add more functions and internal calls if necessary, as long as you never change existing functions. Only files in the include/ and system/ directories should be altered. Do not add any new source files, as this would require altering the build process.

IV. Submission

You should submit your *main*() program along with your syscall implementations. Also be sure to include any Xinu headers/sources that you modified, with comments on what you have added - everything should be packaged into a .zip file. For grading, the TAs will use their own *main()* function to compile and test the functionality of your system calls, in addition, of course, to inspecting your code itself.