CS 503 - Fall 2014

Lab 3
Process Signalling
Extended Message Passing

Due Sunday, September 28th, 2014 at 11:59 PM

Objectives:

By the end of this lab you will be able to:

- Understand processes in XINU can communicate with each other using messages
- Extend the messaging interface to allow multiple messages to be sent to a single process without any message loss.

Background

The XINU process table consists of an entry for every process in the system. The process table keeps track of the process status, priority, stack pointer, parent, etc. (see include/process.h for a full description). Processes can communicate with each other in several ways (global variables, semaphores, etc.). One such method of communication is the send/receive set of system calls. These consist of the following calls:

- send - send a message to a process
- receive - wait for a message to be sent from another 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
- recvtime - performs the same behavior as receive, yet will only wait for a specified time for the message to be received

In the case of send, if a process has already been sent a message, the send system call returns SYSERR and does not deliver the message to the target process.

In this lab, your job is to add new system calls to allow for multiple messages to be sent to and received by a single target process. You will be implementing the following system calls:

```
/* Send a message to the targeet process, allowing multiple messages */
syscall mysend(pid32 pid, umsg32 msg);

/* Receive a message which was sent using mysend */
umsg32 myreceive(void);
```

```
/* Send multiple messages to a process */
uint32 mysendn(pid32, umsg32* msgs, uint32 msg_count);

/* Receive multiple messages */
syscall myreceiven(umsg32* msgs, uint32 msg_count);
```

Setup

In /homes/cs503/xinu there is a file called xinu-fall2014-lab3.tar.gz that contains a start to the code. Unpack:

```
tar zxvf /u/u3/cs503/xinu/xinu-fall2014-lab3.tar.gz
```

This will create a directory called xinu-fall2014-lab3.

Along with the main code for XINU, this tarball contains the following files (additional explanation of the contents of the files are in the following sections).

- system/mysend.c - function declarations for the mysend and mysendn system calls.
- system/myreceive.c - function declarations for the myreceive and myreceiven system calls.
- include/prototypes.h - prototypes for the new system calls.

What's in system/mysend.c

The file mysend.c file contains the function declarations for the mysend and mysendn functions that you must implement:

- syscall mysend(pid32 pid, umsg32 msg) - System call for the mysend function. This system call sends the message (msg) to the process identified by pid. If the process (pid) already has a message waiting, the new message is queued. The function returns OK on success or SYSERR on error.
- uint32 mysendn(pid32, umsg32* msgs, uint32 msg_count) - System call for the mysendn function. This system call sends msg_count messages from the msgs array to the given process (pid). It returns the number of messages actually sent or SYSERR on error.

You are required implement the bodies of these functions.

What's in system/myreceive.c

The file myreceive.c file contains the function declarations for the myreceive and myreceiven functions that you must implement:

- umsg32 myreceive(void); - System call for the myreceive function. This system call causes the calling process to wait for a message to be sent to it using mysend or mysendn. If a message has already been sent it is immediately received.

- syscall myreceiven(umsg32* msgs, uint32 msg_count) - System call for the myreceiven function. This system call causes the calling process to wait for msg_count messages to be sent to it using mysend or mysendn. If enough messages are in the process's message queue, they are immediately received.

You are required implement the bodies of these functions.

Additional Requirements:

- With mysend and mysendn, the process identifier (pid) must be the identifier for a valid process. If a process identifier is passed to mysend or mysend that is not valid, SYSERR should be returned. See an example of how to do this by looking at the send system call in send.c
- Messages should be received in the order in which they were sent. You will need to alter the process table entry to make sure sent messages are queued and dequeued correctly.
  HINT: Consider using a circular buffer as you did in lab1.
  NOTE: Use disable to disable interrupts at the beginning of the system call and the restore system call and enable to re-enable interrupts prior to returning. This will allow exclusive access to the process table. Do NOT use a semaphore. Use the existing send and receive system calls as an example.
- Do not change the way that the existing send, receive, and recvclr system calls work. You will be required to change the process table entry for the new system calls, but do not change it in away that causes existing system calls to no longer work. A call to send, receive, recvclr, and recvtime should function in the same way they do today.
- In order to not affect the behavior of existing system calls, you are required to add a new process state (MYRECV). A process should be set to this state anytime it is waiting for a message from a call to myreceive or myreceiven. TIP: You can set this state in a similar to that receive sets the process state, see receive.c. Process states are defined in include/process.h
- While we'd like to be able to mysend an unlimited number of messages to a process, we realize that in operating systems, nothing is limitless. So you will only have to support a maximum of 20 messages. If a process has received 20 messages and another process tries to send it a message using mysend, then return SYSERR just as send does today.
- The mysendn system call always returns the number of messages actually sent to the process or SYSERR if the pid is not valid. If a process can not receive all the messages from a call to mysendn due to a full message queue, send as many messages as possible to the process and return the number that were actually sent, including zero if the queue is completely full when mysendn is called.
- The myreceiven system call should be as efficient as possible. You are not allowed to use a while loop to determine if all the messages are ready to be received. Instead, keep track of the total number of messages that the process wants to receive and only put it on the ready queue when it has been sent the correct number of messages.
- Provide a set of test cases to ensure that your code works as required. Put these test cases in main.c
- NOTE: When you make xinu for this lab the make file will generate two files in the compile directory:.
  - xinu - this is the file you will download to the xinu backend (you will NOT use xinu.xbin)
  - xinu.elf - this is the executable and linkable format version of the xinu binary. This is not the format to use when sending to the backends. It is useful for low level (assembly) debugging only.

Extra Credit:

Implement the myrecvtime which behaves similar to the recvtime system call:

```
uint32  myrecvtime(int32 maxwait, umsg32* msgs, uint32 msg_count);
```

The myrecvtime system call blocks and returns if one of the two conditions have been met:

- The number of messages specified in msg_count have been received by the process from either mysend or mysendn.
- A timeout of maxwait has occured

In the second case (the case of the timeout), any message that have already been sent (using mysend or mysendn) are returned in the msgs array. In all cases the number of messages actually received are returned by the myrecvtime function.

Some additional requirements and tips:

- You are not allowed to have myrecvtime loop until it has received enough messages. The process receiving process should only be placed on the ready queue if the timeout has elapsed or if the correct number of messages have been received.
- Do not modify the behavior of any other existing system call.
- Feel free to add any additional process state that you need, but keep in mind not to change the behavior of any other system call.

What to turn in

Submit using turnin command your complete source code (all of XINU) including the any files you added to complete the lab. In the 'system' directory include a PDF file with a write up discussing:

- The details behind your implementation. As part of this discussion write answers to the following questions:
  - How does your solution guarantee messages are received in the order in which they were sent?
  - Describe your test cases. How to they ensure that the system calls correctly meet the requirements?
  - What modifications would need to be made to allow for a truly unlimited number of messages to be sent to a target process? Are these modifications practical?

To turn in your lab use the following command

```
turnin -c cs503 -p lab3 xinu-fall2014-lab3
```

assuming xinu-fall2014-lab4 is the name of the directory containing your code.

If you wish to, you can verify your submission by typing the following command:

```
turnin -v -c cs503 -p lab3
```

Do not forget the -v above, as otherwise your earlier submission will be erased (it is overwritten by a blank submission).

Note that resubmitting overwrites any earlier submission and erases any record of the date/time of any such earlier submission.

We will check that the submission timestamp is before the due date; we will not accept your submission if its timestamp is after the due date. Do NOT submit after 11:59 PM Eastern Time.