**CS 354 Spring 2018**

**Lab 5: Enhancing Kernel IPC Services: Blocking and Asynchronous IPC (330 pts)**

**Due: 04/03/2018 (Tue.), 11:59 PM**

**1. Objectives**

The objective of this lab is to enhance XINU's kernel services so that blocking IPC in the form of 1-word message blocked transmission and asynchronous IPC using callback function are added.

**2. Readings**

Read Chapter 8 of the XINU textbook.

**3. Blocking message send [130 pts]**

This problem concerns the implementation of a blocking version of send(), call it sendblk(). sendblk() follows the same function definition as send(). If the receiver's 1-word message buffer is empty, sendblk() behaves like send(). If the receiver's buffer is full, sendblk() blocks indefinitely. Upon blocking, XINU's upper half context switches out the sending process and puts it into a waiting state SNDBLK. This is implemented by defining an additional constant PR\_SNDBLK in process.h with value 20. The 1-word message is stored in the sending process's process table entry where three new fields are defined:

bool8 sendblkflag; /\* Nonzero iff blocking to send \*/

umsg32 sendblkmsg; /\* Message attempting to send \*/

pid32 sendblkrcp; /\* PID of recipient \*/

sendblkflag is nonzero if sendblkmsg is valid, sendblkmsg holds the 1-word message to be sent, and sendblkrcp is the PID of the receiver process.

We introduce two additional fields in the process table structure procent

bool8 rcpblkflag; /\* Nonzero iff one or more process are waiting to send \*/

qid16 sendqueue; /\* Index to FIFO queue of blocked senders \*/

where rcpblkflag indicates whether there are blocked sending processes and sendqueue is an index to the FIFO queue of blocked sending processes. A blocking sender process is put on the per-receiver FIFO waiting queue pointed to by sendqueue. At the receiver end, before a receiving process returns from receive() it must check if there are any blocked sending processes. If there are, unblock the first process in the FIFO sendqueue which includes copying the message from the sender's message buffer sendblkmsg to the receiver's buffer prmsg and setting the appropriate flags.

Implement sendblk() using the fixed priority scheduler of XINU. Create test cases whose output demonstrate the correct functioning of the blocking message send extension of kernel services. Include in your test cases sluggish receivers that sleep so that its FIFO sender queue builds up. In Lab5Answers.pdf describe how you set up your test cases and how your verified correctness of your implementation. send() must work as is without being impacted by sendblk(). Although there is no need to implement, describe what you think should happen if a receiver process terminates when one or more processes are blocked in its sendqueue. Explain the rationale behind your design decision. As in previous labs, please save each C function in a separate file following the function name (e.g., sendblk.c for sendblk()) and place them in system/.

**4. Asynchronous IPC with callback function (200 pts)**

This problem concerns implementation of asynchronous IPC with callback function in XINU as discussed in class. A process registers a user callback function with the kernel requesting that it be executed on behalf of the process when a 1-word message arrives. The callback function contains code written by the an app programmer. Although XINU does not implement isolation/protection, we will adopt a design suitable for modern operating systems such as Linux/UNIX and Windows that must ensure that isolation/protection is preserved when a kernel provides this service.

When a 1-word message is sent to the receiver process that registered a callback function -- in our uni-processor backends this implies that the sender process is the current process running on the CPU -- XINU will not run the callback function as kernel. That is, as part of upper or lower half code running in kernel mode in Linux/UNIX and Windows (in XINU everything runs in kernel mode). Nor will XINU borrow the sender process's context to run the callback function since processes must be shielded from each other. Instead, the kernel waits until its scheduler context-switches in the receiver process. Before resuming the receiver process's execution where it left off, XINU will inject a jump to the registered callback function so that the callback function is executed in the context of the process that registered it. After the callback function returns, the receiver process resumes where it originally left off. Thus the receiver process's app code does not call the callback function directly. Beyond initially registering the callback function, the user process has no involvement in the eventual scheduling and execution of the callback function. Instead the kernel, in our case XINU, sets up the function call at run-time prompted by a message send event.

The kernel exports a system call, cbreg(), that is used by an application to specify a user space callback function to be executed in its context when a message is successfully received. By "successfully" we mean that the receiver process's message buffer was empty so that the message could be saved. Otherwise, message transmission was unsuccessful and the callback function is not invoked. The function definition is given by

* syscall cbreg( int (\* fnp) (void) )

where fnp is a function pointer to a user space callback function. For simplicity, a callback function returns a value of type int32 and takes no argument. What the callback function does is up to the app programmer. An example callback function is

int32 mrecv\_cb(void) {

extern umsg32 mbuf;

mbuf = receive();

return(OK);

}

where mbuf is a global variable in the receiver process's memory space to which the received message is copied. Hence it is the responsibility of the callback function to copy the message from kernel buffer to user buffer. Note that although receive() is a blocking system call, it will not block since mrecv\_cb() will be invoked only after a message has been received. A XINU process (e.g., the process running main()) registers a callback function using

if (cbreg(&mrecv\_cb) != OK) {

kprintf("cb registration failed!");

return 1;

}

XINU remembers that a callback function has been registered by introducing two new fields in the process table structure procent:

bool8 prhascb; /\* Nonzero iff callback function has been registered \*/

int (\* fptr) (); /\* Pointer to cb function if one has been registered \*/

prhascb is a Boolean flag and fptr is NULL if no callback function has been registered. When a process tries to register a second callback function, cbreg() should return with an error.

To implement asynchronous IPC in XINU whose design is compatible with kernels that support isolation/protection, XINU runs the receiver's callback function in the context of receiver process when it is scheduled next. Use a XINU version that uses the fixed priority scheduler as in Problem 3. Describe your design that allows XINU to implement asynchronous IPC with callback function in Lab5Answers.pdf. Discuss why your design is compatible with kernels that support isolation/protection even though it is implemented in XINU. Test your design and implementation using sender and receiver processes. This includes sending of multiple messages by a single sender and multiple sender processes. Use kprintf() to track the relevant events to help establish correctness. Save new functions in separate files under system/ following our naming convention.

**Bonus problem [20 pts]**

Using the man pages of Linux msgsnd() and msgrcv() as reference examples, write man pages for receive() and sendblk() as plain ASCII text files (man pages follow the troff typesetting format) receive.2 and sendblk.2 in system/. Make the man pages as unambiguous and complete as feasible.

**Turn-in Instructions**

Before submitting your work, make sure to double-check the [TA Notes](http://www.cs.purdue.edu/homes/ogg/cs354_spring2018.html) to ensure that additional requirements and instructions have been followed.

*Electronic turn-in instructions:*

        i) Go to the xinu-spring2018/compile directory and do "make clean".

        ii) Go to the directory of which your xinu-spring2018 directory is a subdirectory. (NOTE: please do not rename xinu-spring2018, or any of its subdirectories.)

                e.g., if /homes/joe/xinu-spring2018 is your directory structure, go to /homes/joe

        iii) Type the following command

                turnin -c cs354 -p lab5 xinu-spring2018

You can check/list the submitted files using   
  
turnin -c cs354 -p lab5 -v

***Important: Please provide comments inside your code so that its function and flow can be conveyed to the reader. Turn off all debugging output before you submit your code.***