CS 503 Lab 3

Process Signalling
Extended Message Passing

Shuheng Guo

1. Introduction

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. 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.

The objective of this lab is as follows:

- 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.

So in this lab, I am going to add new system calls to allow for multiple messages to be sent to and received by a single target process. I will implement 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);
```

For extra credit, there is one more function:

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

2. Implementation and Explanation

2.1. Process.h

In order to remain the existing send, receive, and recvclr system calls unchanged and still working in their way, I made some change in process.h.

```
/* Process state constants */

#define PR_FREE         0       /* Process table entry is unused        */
#define PR_CURR         1       /* Process is currently running         */
#define PR_READY        2       /* Process is on ready queue            */
#define PR_RECV         3       /* Process waiting for message          */
#define PR_SLEEP        4       /* Process is sleeping                  */
#define PR_SUSP         5       /* Process is suspended                 */
#define PR_WAIT         6       /* Process is on semaphore queue        */
#define PR_RECTIM       7       /* Process is receiving with timeout    */
#define PR_MYRECV       8       /* Process is waiting for a queued msg  */
#define PR_MYRECTIM     9   /* Process is receiving queued msg with timeout*/
```

Figure 1 process state constants

In this definition of process constants, the last two constants are what I am going to use in this lab. They are used to specify process waiting for a queued message with/without timeout respectively.

```
/* Definition of the process table (multiple of 32 bits) */

struct procent {                /* Entry in the process table           */
        uint16  prstate;        /* Process state: PR_CURR, etc.         */
        pri16   prprio;         /* Process priority                     */
        char    *prstkptr;      /* Saved stack pointer                  */
        char    *prstkbase;     /* Base of run time stack               */
        uint32  prstklen;       /* Stack length in bytes                */
        char    prname[PNMLEN]; /* Process name                         */
        uint32  prsem;          /* Semaphore on which process waits     */
        pid32   prparent;       /* ID of the creating process           */
        umsg32  prmsg;          /* Message sent to this process         */
        bool8   prhasmsg;       /* Nonzero iff msg is valid             */
        int16   prdesc[NDESC];  /* Device descriptors for process       */
        int32   cap;            /* queue capacity                       */
        int     message[20];    /* msg queue                            */
        int32   head;           /* head of queue                        */
        int32   tail;           /* tail of queue                        */
        int32   count;          /* number of msg in queue               */
};
```

Figure 2 process table

I added five new entries to the process table. There is an array of integer to function as a message queue. The relative variables cap, head, tail, count are used to realize enqueue and dequeue functions in the following.

2.2 Enqueue and Dequeue

```
void enqueueue(pid32 pid, umsg32 item){
        struct procent *prptr = &proctab[pid];
        prptr->message[prptr->tail] = item;
        prptr->tail = (prptr->tail + 1) % 20;
        prptr->count++;
        prptr->cap--;
}
```

Figure 3. enqueue

Function enqueueue is to add a message to recipient's queue. If multiple messages are going to be sent, there will be a loop.

```
umsg32 dequeueue() {
        struct procent *prptr = &proctab[currpid];
        umsg32 item = prptr->message[prptr->head];
        prptr->head = (prptr->head + 1) % 20;
        prptr->count--;
        prptr->cap++;
        return item;
}
```

Figure 4. dequeue

Function dequeueue is to remove a message from current process' queue. If multiple messages are going to be received, there will be a loop.

2.3 Mysend.c and Mysendn.c

```
syscall mysend(
          pid32         pid,
          umsg32        msg
        )
{
    intmask mask;
    struct procent *prptr;

    mask = disable();
    if (isbadpid(pid)) {
        restore(mask);
        return SYSERR;
    }

    prptr = &proctab[pid];
    if (prptr->prstate == PR_FREE || prptr->count >= 20) {
        restore(mask);
        return SYSERR;
    }

    enqueue(pid, msg);

    if (prptr->prstate == PR_MYRECV) {
        ready(pid);
    } else if (prptr->prstate == PR_MYRECTIM) {
        unsleep(pid);
        ready(pid);
    }
    restore(mask);
    return OK;

}
```

Figure 5. Mysend.c

Function mysend.c is to send a single message to recipient. If recipient's message is already full, return error. Otherwise, add a new message to recipient's message queue. If recipient's state is in PR_MYRECV or PR_MYRECTIM, put it to ready list.

```
uint32  mysendn(
          pid32         pid,
          umsg32*       msgs,
          uint32        msg_count
        )
{
    int i;
    int result = msg_count;
    intmask mask;
    struct procent *prptr;

    mask = disable();
    if (isbadpid(pid)) {
        restore(mask);
        return SYSERR;
    }

    prptr = &proctab[pid];
    if (prptr->prstate == PR_FREE) {
        restore(mask);
        return SYSERR;
    }

    if (prptr->cap >= msg_count){
        for (i = 0; i < msg_count; i++) {
            enqueueue(pid, *(msgs+i));
        }
    } else if(prptr->cap >= 0 && prptr->cap < msg_count) {
        for (i = 0; i < prptr->cap; i++){
            enqueueue(pid, *(msgs+i));
        }
        result = prptr->cap;
    }


    if (prptr->prstate == PR_MYRECV) {
        ready(pid);
    } else if (prptr->prstate == PR_MYRECTIM) {
        unsleep(pid);
        ready(pid);
    }
    restore(mask);

    return result;


}
```

Figure 6. Mysendn.c

Function mysendn.c is to send multiple messages to recipient. Return the number of messages actually sent.

2.4 Myreceive.c and Myreceiven.c

```
umsg32  myreceive(void)
{
    intmask     mask;
    struct      procent *prptr;
    umsg32      msg;

    mask = disable();
    prptr = &proctab[currpid];

    if (prptr->count <= 0) {
        prptr->prstate = PR_MYRECV;
        resched();
    }

    msg = dequeueue();
    restore(mask);
    return msg;

}
```

Figure 7 Myreceive.c

Function myreceive is to receive and return a message if it is already in queue. Otherwise, put process' state to PR_MYRECV and reschedule.

```
syscall myreceiven(
          umsg32*       msgs,
          uint32        msg_count
        )
{
    intmask     mask;
    struct      procent *prptr;
        int i;
        int mes;

    mask = disable();
    prptr = &proctab[currpid];

    while (prptr->count < msg_count) {
        prptr->prstate = PR_MYRECV;
        resched();
    }

        for (i = 0; i < msg_count; i++){
                mes = dequeueue();
                *(msgs + i) = mes;
        }
    restore(mask);
    return OK;

}
```

Figure 8. Myreceiven.c

Function myreceiven is to receive and return multiple messages if there are enough messages already in queue. Otherwise, put process' state to PR_MYRECV and reschedule.

2.5 Extra Myrecvtime.c

```
umsg32  myrecvtime(
          int32         maxwait,
          umsg32*       msgs,

          uint32        msg_count
        )
{
        intmask mask;
        struct  procent *prptr;

        if (maxwait < 0) {
                return SYSERR;
        }
        mask = disable();

        prptr = &proctab[currpid];
        if (prptr->count == 0) {
                if (insertd(currpid,sleepq,maxwait) == SYSERR) {
                        restore(mask);
                        return SYSERR;
                }
                prptr->prstate = PR_MYRECTIM;
                resched();
        }

        int msg = 0;
        while(prptr->count > 0) {
                msgs[msg] = dequeueue();
                msg++;
                if( msg == msg_count) {
                        break;
                }
        }
        restore(mask);
        return msg;
}
```

Figure 9 Myrecvtime.c

Function myrecvtime allow the process to wait for a certain time and return the number of messages actually received.

3. Results and Analysis

To measure the functionality of all the functions I have written, I wrote three test cases to examine mysend/myreceive, mysendn/myreceiven, myrecvtime respectively.

```
void cocosend1(pid32 pid, umsg32 msg){
                wait(general);
                mysend(pid, msg);
}

void cocosend2(pid32 pid, umsg32* msgs, uint32 msend){
                wait(general);
                mysendn(pid, msgs, msend);
}

void cocoreceive1(){
                wait(general);
                int mes = myreceive();
                kprintf("The message I received is %d\n", mes);
                kprintf("\n");
}

void cocoreceive2(umsg32* new_msgs, uint32 mreceive){
                wait(general);
                int i;
                myreceiven(new_msgs,mreceive);
                for(i = 0; i < mreceive; i++){
                        kprintf("The message I received is %d\n", *(new_msgs+i));
                }
                kprintf("\n");
}

void cocorecvtime( int32 maxwait, umsg32* msgs, uint32 msg_count){
                wait(general);
                int num;
                int i = 0;
                num = myrecvtime(maxwait, msgs, msg_count);
                kprintf("there are %d messages received. And these messages are:\n", num);
                for(; i < num; i++) {
                        kprintf(" %d ", *(msgs + i));
                }
                kprintf("\n");
}
```

Figure 10 Test Case

And the results are:

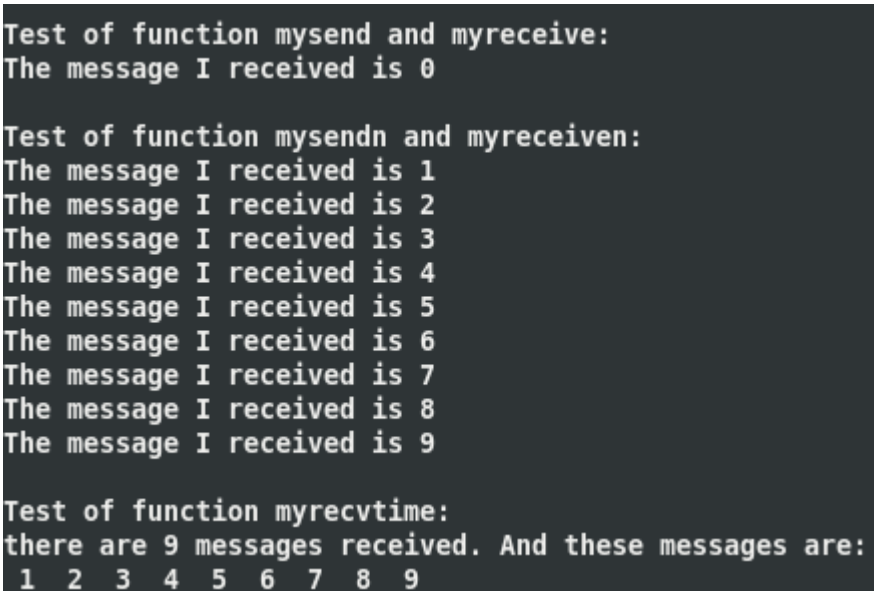

Figure 11 Lab3 Results

From Figure 11 we can see that:

- In case 1: Process execute myreceive function received the single message from process which executes mysend function and the message is an integer with value 0.
- In case 2: Process execute myreceiven function received the multiple messages from process which executes mysendn function and the messages are an array of integers with value from 1 to 9.
- In case 3: Process execute myrecvtime function received the multiple messages from process which executes mysendn function within the maxwait time. And in this case, the messages are an array of integers with value from 1 to 9 and maxwait time is 1000ms.

The results are reasonable and accessible. That means the functions I wrote in lab3 match with our initial objectives.

4. Discussion

- Q: How does your solution guarantee messages are received in the order in which they were sent?
- A: All messages have to go through a FIFO queue to be received. All the messages obey the first in first out rule so it is guaranteed that messages are received in the order they were sent.
- Q:Describe your test cases. How do they ensure that the system calls correctly meet the requirements?
- A: In the test case, I created two processes to perform as sender and recipient respectively each time. The master program sleep a short time and than begin another test case.
- Q :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?
- A: To send unlimited messages I suggest to use dynamic array instead of the fixed size array as message queue. Whenever the queue is full, just double the size of queue to allow more messages get in the queue.