Lab 6

CIS-657 PRINCIPLES OF OPERATING SYSTEMS

PROF. MOHAMMED ABDALLAH

3/9/2021

Team 4

Anthony Redamonti
Adrian Bernat
Dana Dippery
Michael Rice
Ousmane Barry

Process.h

The following changes were made to process.h to implement the message buffer. First, the number of messages per process were defined as 5 in NMSG.

```
/* Number of messages per process */
#define NMSG        5
```

Then to implement the buffer, the prmsg was changed to an array of type umsg32. The prhasmsg flag was removed and replaced with the int32 “msgindex” variable.

```
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[NMSG];        /* messages sent to this process     */
    int32   msgindex;
    int16   prdesc[NDESC];  /* device descriptors for process   */
};
```

Create.c

The following lines were added to create.c to initialize the new parameters when a process is created. The msgindex can be thought of as the number of messages a process has received. It is initialized with the value 0.

```
    prptr->msgindex = 0;
    for (i=0 ; i<NMSG; i++){
        prptr->prmsg[i] = NULL;
    }
```

Send.c

First, the original send.c function was changed to implement the new parameters. If the state of the process is free or if the number of messages already in the buffer equals NMSG (buffer is full) then the SYSERR is returned. If neither of those conditions are true, then the message is allocated in a slot in the buffer and the msgindex is incremented by one.

```
    prptr = &proctab[pid];
    if ((prptr->prstate == PR_FREE) || prptr->msgindex == NMSG) {
```

```
        restore(mask);
        return SYSERR;
    }
    prptr->prmsg[prptr->msgindex] = msg;        /* deliver message      */
    prptr->msgindex++;      /* increment index for next message */
```

Receive.c

The original receive.c function was changed to implement the new parameters. If no messages are present (msgindex == 0), then the process state is set to PR_RECV, and resched() is called. Otherwise, the message is copied from the most recent message in the buffer and the msgindex is decremented.

```
    prptr = &proctab[currpid];
    if (prptr->msgindex == 0) {
        prptr->prstate = PR_RECV;
        resched();      /* block until message arrives  */
    }
    msg = prptr->prmsg[prptr->msgindex-1];      /* retrieve message     */
    prptr->msgindex--;  /* reset message flag       */
```

Recvclr.c

The original recvclr.c function was edited to implement the new parameters. If a message was present (msgindex > 0) then the message was copied from the most recent message in the buffer and the msgindex was decremented.

```
    prptr = &proctab[currpid];
    if (prptr->msgindex > 0) {
        msg = prptr->prmsg[prptr->msgindex-1];  /* retrieve message     */
        prptr->msgindex--;
    } else {
        msg = OK;
    }
```

Recvtime.c

The original recvtime.c function was changed to implement the new parameters. If the msgindex <= 0 (no message in buffer) then the process is inserted into the sleeping queue with a state of PR_RECTIM and resched() is called. If there is a message (msgindex > 0) then it was copied from the most recent message in the buffer and the msgindex was decremented. If there was not a message, then the message returned would be “TIMEOUT”.

```
    prptr = &proctab[currpid];
    if (prptr->msgindex <= 0) { /* if message waiting, no delay */
        if (insertd(currpid,sleepq,maxwait) == SYSERR) {
```

```
            restore(mask);
            return SYSERR;
        }
        sltop = &queuetab[firstid(sleepq)].qkey;
        slnonempty = TRUE;
        prptr->prstate = PR_RECTIM;
        resched();
    }

    /* Either message arrived or timer expired */

    if (prptr->msgindex > 0) {
        msg = prptr->prmsg[prptr->msgindex];    /* retrieve message     */
        prptr->msgindex--; /* reset message indicator   */
    } else {
        msg = TIMEOUT;
    }
```

Sendk.c

Sendk.c takes three arguments: the pid of the recipient process, the msg array to send to the recipient, and the size of the message array. If the state of the recipient process is PR_FREE, SYSERR is returned. In a for-loop, each message in the msg argument is loaded into a free slot in the prmsg parameter. If the buffer is full and no messages were sent (first iteration of for-loop), an error message is printed “ERROR: Buffer is full” and SYSERR is returned. If the buffer is full but some of the messages were sent, then an error message is printed “ERROR: Buffer is full”. Finally, if the recipient process was waiting on a message, it will be made ready and resched() will be called. Also, if the recipient process was in a timed wait, unsleep() will remove the process from the sleeping list, ready() will make it ready and resched() will be called.

```
/* sendk.c - sendk */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  sendk  -  pass multiple messages to a process and start recipient if waiting
 *------------------------------------------------------------------------
 */
syscall sendk(
      pid32     pid,        /* ID of recipient process  */
      umsg32    msg[],      /* contents of message      */
      uint32    msgsize     /* size of the msg array    */
    )
{
    int32 i;
```

```
    intmask mask;           /* saved interrupt mask     */
    struct  procent *prptr;     /* ptr to process' table entry  */

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

    prptr = &proctab[pid];
    if (prptr->prstate == PR_FREE) {
        restore(mask);
        return SYSERR;
    }
    for(i=0; i<msgsize;i++){
        if(prptr->msgindex == NMSG){
            if(i == 0){
                restore(mask);
                kprintf("ERROR: Buffer is full\n");
                return(SYSERR);
            }
            else{
                kprintf("ERROR: Buffer is full\n");
                break;
            }
        }
        else{
            prptr->prmsg[prptr->msgindex] = msg[i];
            prptr->msgindex++;      /* increment index for next message */
        }
    }

    /* If recipient waiting or in timed-wait make it ready */

    if (prptr->prstate == PR_RECV) {
        ready(pid, RESCHED_YES);
    } else if (prptr->prstate == PR_RECTIM) {
        unsleep(pid);
        ready(pid, RESCHED_YES);
    }
    restore(mask);      /* restore interrupts */
    return OK;
}
```

Receivek.c

Receivek.c takes the number of messages as its argument. The number of messages is used to determine the size of the buffer to allocate in memory. The uint32 “bytesneeded” variable is equal to 8 multiplied by the number of messages to receive. The buffer is allocated to “umsg32* msg” using the getmem function. In a for-loop the messages are received and copied into the msg buffer and the msgindex is decremented. If the number of messages received is zero, then the process state will be PR_RECV and will call resched(). Finally, the msg buffer is returned.

```
/* receivek.c - receivek */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  receivek  -  wait for multiple messages and return the messages to the caller
 *------------------------------------------------------------------------
 */
umsg32* receivek(uint32 numofmsgs)
{
    intmask mask;           /* saved interrupt mask     */
    struct  procent *prptr;     /* ptr to process' table entry  */
    umsg32* msg;            /* message to return        */
    int32 i;
    uint32 bytesneeded = 8 * numofmsgs;

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

    msg = (umsg32*)getmem(bytesneeded);

    for(i = 0; i < numofmsgs; i++){
        if (prptr->msgindex == 0) {
            prptr->prstate = PR_RECV;
            resched();      /* block until all messages arrive  */
        }
        msg[i] = prptr->prmsg[prptr->msgindex-1];  /* retrieve message     */
        if(prptr->msgindex > 0){
            prptr->msgindex--
;  /* if messages present, decrement message index for next message    */
        }
    }
    restore(mask);
    return msg;
}
```

Main.c

Main.c creates two processes “m1” and “m2” and resumes them. M1 has a priority of 40 and m2 has a priority of 20. M1 will print 10 messages and send all of them to m2 using the sendk function. M2 will receive 5 messages from m1 using the receivek function, print the messages received, then free the buffer using the freemem function.

```
/* main.c - main */
#include <xinu.h>
void m1();void m2();
pid32 m1pid, m2pid;

void main(void)
{
    m1pid=create(m1, 1024, 40,"m1",0);
    m2pid=create(m2, 1024, 20,"m2",0);
    resume(m1pid);
    resume(m2pid);
    return OK;
}

void m1(){
    uint32 i;
    umsg32 message[10] = {1,2,3,4,5,6,7,8,9,10};
    kprintf("contents of array before sending: ");
    for(i = 0; i < 10; i++){
        kprintf("%d ", message[i]);
    }
    kprintf("\n");
    sendk(m2pid, message, 10);
}

void m2(){
    int32 i;
    umsg32 *message;
    kprintf("contents of array received: ");
    message = receivek(5);
    for(i = 0; i < 5; i++){
        kprintf("%d ", message[i]);
    }
    freemem(message, 40);
}
```

Output

Below is the output of the minicom. M1 prints the contents of the 10 messages sent to m2. Since the supported maximum size of the buffer is 5, an error message is generated following an attempt to send a sixth message. M2 prints out the content of the messages received. The messages are received in LIFO order, so the order of the messages received is 5, 4, 3, 2, 1.