Final Exam

CIS-657 PRINCIPLES OF OPERATING SYSTEMS

PROF. MOHAMMED ABDALLAH

3/22/2021

Anthony Redamonti
SYRACUSE UNIVERSITY

Question 1: resched.c

The edits to resched.c are highlighted in blue below. “Pupdate” is a pointer to a process entry used to update the priority of starving processes on the ready list. “Ptfirstready” is a pointer to the first process in the ready list. The integers “walkready” and “secondready” store indexes in the queuetab and are used to walk the ready list.

There are two while loops which handle two different scenarios. The first while loop handles the scenario of these three conditions:

- Ptold points to the current process.
- The priority of the current process is greater than the priority of the first ready process.
- The quantum has expired.

In this case, all the priorities of the ready processes are incremented by one, excluding the NULL process. If the priority of the current process is greater than the priority of the first process, but the quantum has not expired, resched returns.

The second while loop handles all other scenarios. If the number of ready process is greater than one, excluding the NULL process, the priorities of the ready processes starting at the second ready process to the end of the ready list (excluding the NULL process) are compared to the priorities of the current process and the first ready process. If they are less than the priorities of the current process and first ready process, they are incremented.

$$if((queuetab[secondready].qkey < ptold \rightarrow prprio)\&\&(queuetab[secondready].qkey < ptfirstready \rightarrow prprio))$$

Note that the qkey is incremented and copied over to the priority of the process stored in the process table. It is important that the priority stored in the process table matches the key stored in the queuetab for each process as the priority and key are compared in the while loops.

```
/* resched.c - resched */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resched  -  Reschedule processor to highest priority eligible process
 *------------------------------------------------------------------------
 */
void    resched(void)       /* assumes interrupts are disabled  */
{
    struct procent *ptold;  /* ptr to table entry for old process   */
    struct procent *ptnew;  /* ptr to table entry for new process   */
    struct procent *pupdate; /* ptr to table entry to update priority */
    struct procent *ptfirstready;/* ptr to table entry for first ready process */
    int16 walkready;      /* used to walk ready list starting at first process */
    int16 secondready;   /* used to walk ready list starting at second process */
```

```
    /* If rescheduling is deferred, record attempt and return */

    if (Defer.ndefers > 0) {
        Defer.attempt = TRUE;
        return;
    }

    /* Point to process table entry for the current (old) process */

    ptold = &proctab[currpid];

    walkready = firstid(readylist);

    if (ptold->prstate == PR_CURR){  /* process remains running */
        if (ptold->prprio > firstkey(readylist)) {
            if(preempt <= 0){ /* increment priority of all ready processes (excep
t NULLPROC) */
                while(queuetab[walkready].qnext != queuetail(readylist)){
                    queuetab[walkready].qkey++;
                    pupdate = &proctab[walkready];
                    pupdate->prprio = queuetab[walkready].qkey;
                    kprintf("PID%d changed priority: %d\n", walkready, pupdate-
>prprio);
                    walkready = queuetab[walkready].qnext;
                }
                preempt = QUANTUM;      /* reset time slice for process */
                return;
            }
            /* do nothing if quantum hasn't expired */
            else{
                return;
            }
        }
    }

    if (ptold->prstate == PR_CURR){  /* process remains running */

    /* Old process will no longer remain current */
    ptold->prstate = PR_READY;
    insert(currpid, readylist, ptold->prprio);
    }

    if(queuetab[walkready].qnext != queuetail(readylist)){ /* if first process in
 ready list not NULLPROC */
```

```
        ptfirstready = &proctab[walkready]; /* pointer to first process in ready
list */
        secondready = queuetab[walkready].qnext;/* PID of second ready process */
        /* while second ready process is not NULLPROC */
        while(queuetab[secondready].qnext != queuetail(readylist)){
                /* The priority of second ready process (and after) must be less
than the priority of the current process and the first ready process if it is to
be incremented. */
            if((queuetab[secondready].qkey < ptold->prprio) &&
(queuetab[secondready].qkey < ptfirstready->prprio)){
                queuetab[secondready].qkey++;
                pupdate = &proctab[secondready];
                pupdate->prprio = queuetab[secondready].qkey;
                kprintf("PID%d changed priority: %d\n", secondready, pupdate-
>prprio);
            }
            secondready = queuetab[secondready].qnext;
        }
    }

    /* Force context switch to highest priority ready process */

    currpid = dequeue(readylist);
    ptnew = &proctab[currpid];
    ptnew->prstate = PR_CURR;
    preempt = QUANTUM;      /* reset time slice for process */
    ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

    /* Old process returns here when resumed */

    return;
}
```

Main.c

Main.c creates and resumes three processes each of which share two global variable integers “count2” and “count3”. The priorities of processes “m1”, “m2”, and “m3” are 18, 15, and 10, respectively. The global variables are initialized at zero and are used to ensure that processes two and three only print their PID once while in the infinite loop.

After the main process terminates, m1 will run first because it has the highest priority of the three processes. In an infinite while loop, it sets both global variables to 0, prints its PID, then sleeps for 2 seconds by calling “sleep(2)”. When m1 sleeps, m2 will run. In an infinite while loop, m2 will set count3

to 0, and if count2 is 0, it will print its PID and set count2 to 1. After 2 seconds has passed, m1 will wake up and call resched(). M3 has the same structure as m2, but it will reset global variable count2 to 0.

```
/* main.c - main */
#include <xinu.h>
void m1();void m2();void m3();
pid32 m1pid, m2pid, m3pid;

int main(void)
{
    int* count2 = (int*)getmem(64);
    int* count3 = (int*)getmem(64);
    count2[0] = 0;
    count3[0] = 0;
    m1pid=create(m1, 1024, 18,"m1",2,count2,count3);
    m2pid=create(m2, 1024, 15,"m2",2,count2,count3);
    m3pid=create(m3, 1024, 10,"m3",2,count2,count3);
    resume(m1pid);
    resume(m2pid);
    resume(m3pid);
    return OK;
}

void m1(int* count2, int* count3){
    while(1){
        count2[0] = 0;
        count3[0] = 0;
        kprintf("RUNNING PID: %d\n", currpid);
        sleep(2);
    }
}

void m2(int* count2, int* count3){
    while(1){
        count3[0] = 0;
        if(count2[0] == 0){
            kprintf("RUNNING PID: %d\n", currpid);
            count2[0] = 1;
        }
    }
}

void m3(int* count2, int* count3){
    while(1){
        count2[0] = 0;
```

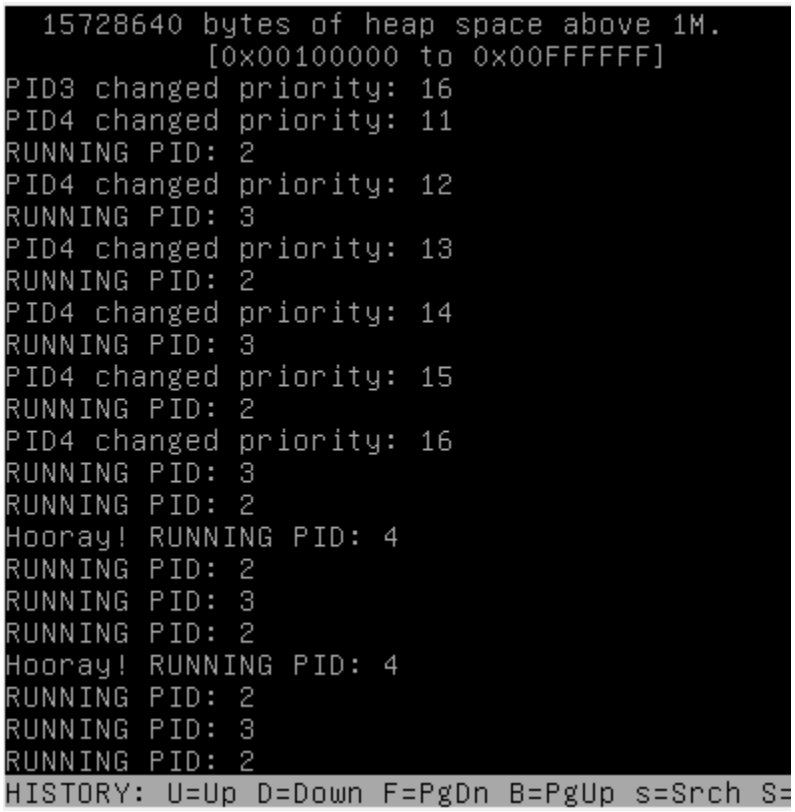
```
        if(count3[0] == 0){
            kprintf("Hooray! RUNNING PID: %d\n", currpid);
            count3[0] = 1;
        }
    }
}
```

Because there is an initial context switch after the main process terminates, the priorities of both m2 and m3 are incremented. Eventually m2 and m3 will have the same priority which will be 16, at which time scheduling will be round robin between m2 and m3.

The expected sequence is as follows (priority increments highlighted in yellow):

RUNNING	READYLIST 1	READYLIST 2	READYLIST 3
Main(20)	m1(18)	m2(15)	m3(10)
m1(18)	m2(16)	m3(11)	
m2(16)	m3(12)		
m1(18)	m2(16)	m3(13)	
m2(16)	m3(14)		
m1(18)	m2(16)	m3(15)	
m2(16)	m3(16)		
m1(18)	m3(16)	m2(16)	
m3(16)	m2(16)		
m1(18)	m2(16)	m3(16)	
m2(16)	m3(16)		
m1(18)	m3(16)	m2(16)	
m3(16)	m2(16)		

The output of the program is below. The results match the expected output.



Question 2: clkinit.S

The edits to clkinit.S are highlighted in yellow below. “Count2” is a variable initialized with a value of 2 and is used to count to two seconds. When a second has passed, count2 is decremented by 1. If count2 equals 0, it is reloaded with the value 2, and checkprio() is called.

```
/* clkint.s - _clkint */

#include <icu.s>
                .text
count2:         .word   2       # count2 used to count 2 seconds
count1000:      .word   1000
                .globl  sltop
                .globl  clkint
                .globl  cl1
clkint:
                pushal
                cli
                movb    $EOI,%al
                outb    %al,$OCW1_2

                incl    ctr1000
                subw    $1,count1000
                ja      cl1
                incl    clktime             # one second has passed. Increment clktime
                movw    $1000,count1000     # reload count1000
                subw    $1,count2           # subtract 1 from count2
                ja              cl1         # if 2 seconds has not passed, do not call checkprio
                movw    $2,count2           # 2 seconds has passed. reload count2
                call    checkprio           # call checkprio
cl1:
                cmpl    $0,slnonempty  # if no sleeping processes,
                je      clpreem        #   skip to preemption check
                movl    sltop,%eax     # decrement key of first
                decl    (%eax)         #   sleeping process
                jg      clpreem        # must use jg for signed int
                call    wakeup         # if zero, call wakeup
clpreem:        decl    preempt        # decrement preemption counter
                jg      clret          # must use jg for signed int
                call    resched        # if preemption, call resched
clret:                                 # return from interrupt
                sti
                popal
                iret
```

Process.h

The edit to process.h is highlighted in blue below. An unsigned 16 bit variable “prdirty” was added to the “procent” struct.

```
struct procent {        /* entry in the process table       */
    uint16  prstate;    /* process state: PR_CURR, etc.     */
    uint16  prdirty;    /* used to indicate whether it ran  */
    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   */
};
```

Create.c

The edit to create.c is highlighted in blue below. When a process is created, the “prdirty” variable is initialized with a value of 0 for that process.

```
    prptr->prdirty = 0;
```

Kernel.h

The edit to kernel.h is highlighted in blue below. The quantum was changed from 2 to 500 to make context switching more visible to the human eye.

```
#define QUANTUM 500     /* time slice in milliseconds       */
```

Resched.c

The edit to resched.c is highlighted in blue below. The prdirty variable is set to 1 indicating that the process has run.

```
    /* Force context switch to highest priority ready process */

    currpid = dequeue(readylist);
    ptnew = &proctab[currpid];
    ptnew->prdirty = 1;
    ptnew->prstate = PR_CURR;
```

```
    preempt = QUANTUM;      /* reset time slice for process */
    ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

    /* Old process returns here when resumed */
```

Checkprio.c

The function checkprio.c was added to the makefile in the compile folder as well as to the prototypes.h header file. Checkprio.c is called every 2 seconds. It iterates through the process table looking for starving processes. If the state of a process is PR_FREE or PR_CURR, it is skipped over. If prdirty is 1, then it is reset to 0 and the process is skipped over. When the priority of a process is incremented, its PID and priority are printed to the console, and the priority is copied to the key value for that process in the queuetab. Also note that the for-loop starts at "i = 1" because the priority of the NULL process is never incremented.

```
/* checkprio.c - checkprio */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  checkprio  -  Called by clock interrupt handler to update priorities of proce
sses.
 *------------------------------------------------------------------------
 */
void    checkprio(void)
{
    struct  procent *prptr;     /* pointer to process       */
    int32   i;          /* index into process table     */
    for (i = 1; i < NPROC; i++) {
        prptr = &proctab[i];
        /* skip unused slots. Do not increment priority of current process */
        if ((prptr->prstate == PR_FREE)||(prptr->prstate == PR_CURR)) {
            continue;
        }
        /* if process has been current in last 2 seconds, do not increment priori
ty. */
        if(prptr->prdirty == 1){
            prptr->prdirty = 0; /* reset dirty flag */
            continue;
        }
        prptr->prprio++; /* increment priority in process table */
        queuetab[i].qkey = prptr-
>prprio; /* copy priority to key of process in queuetab */
        kprintf("PID%d changed priority: %d\n", i, prptr->prprio);
    }
    return;
```

```
}
```

Main.c

Main.c creates and resumes three processes each of which share three global variable integers count1, count2, and count3. The priorities of processes “m1”, “m2”, and “m3” are 18, 15, and 10, respectively. The global variables are initialized with a value 0 and are used to ensure that each process prints its PID only once while in the infinite loop.

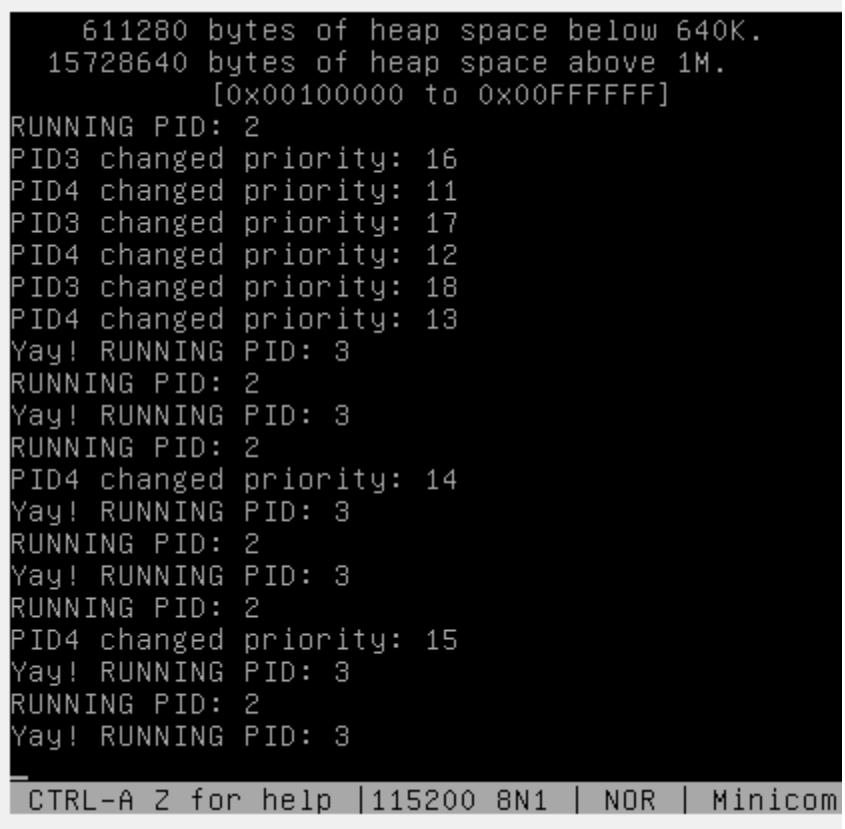
After the main process terminates, m1 will run first because it has the highest priority of the three processes. In an infinite while loop, it sets global variables count2 and count 3 to 0, and if count1 equals 0, it prints its PID and sets count1 to 1. M2 will starve until its priority matches the priority of m1. Once checkprio() increments m2 three times, its priority will match m1, and it will run. Because checkprio() runs every 2 seconds, m2 will be starved for 6 seconds (3*2sec = 6sec). Using the same logic, m3 will run after 16 seconds (8*2sec = 16 sec).

In an infinite while loop, m2 will set count1 and count3 to 0, and if count2 is 0, it will print its PID and set count2 to 1. M3 has the same structure as m2, but it will reset global variables count1 and count2 to 0.

```
/* main.c - main */
#include <xinu.h>
void m1();void m2();void m3();
pid32 m1pid, m2pid, m3pid;

int main(void)
{
    int* count1 = (int*)getmem(64);
    int* count2 = (int*)getmem(64);
    int* count3 = (int*)getmem(64);
    count1[0] = 0;
    count2[0] = 0;
    count3[0] = 0;
    m1pid=create(m1, 1024, 18,"m1",3,count1,count2,count3);
    m2pid=create(m2, 1024, 15,"m2",3,count1,count2,count3);
    m3pid=create(m3, 1024, 10,"m3",3,count1,count2,count3);
    resume(m1pid);
    resume(m2pid);
    resume(m3pid);
    return OK;
}

void m1(int* count1, int* count2, int* count3){
    while(1){
        count2[0] = 0;
        count3[0] = 0;
```

```
        if(count1[0] == 0){
            count1[0] = 1;
            kprintf("RUNNING PID: %d\n", currpid);
        }
    }
}

void m2(int* count1, int* count2, int* count3){
    while(1){
        count1[0] = 0;
        count3[0] = 0;
        if(count2[0] == 0){
            count2[0] = 1;
            kprintf("Yay! RUNNING PID: %d\n", currpid);
        }
    }
}

void m3(int* count1, int* count2, int* count3){
    while(1){
        count1[0] = 0;
        count2[0] = 0;
        if(count3[0] == 0){
            count3[0] = 1;
            kprintf("Yay!! RUNNING PID: %d\n", currpid);
        }
    }
}
```

The output of the program is below. Once m1 runs for 2 seconds, checkprio() increments the priorities of m2 and m3. Once 6 seconds passed, the priority of m2 matched m1, so m2 printed its PID.



Once 16 seconds passed, m3 had a priority of 18 as well, so it joined in the round-robin scheduling with m1 and m2.