

Solutions to Exam One

CS160-401 - Operating Systems Drake University - Spring, 2002

Directions: Do five out of the following six problems. You must cross out the problem that you do not wish graded. Show all work. Please work first on problems with which you are most comfortable.

Problem 1 Assume that N is a positive integer constant. Write some code that will result in N processes executing, with the original process creating one child process, and this process creating one child process, and this process creating one child process, and so on. Have each of the first $N - 1$ processes wait on its own child process's termination before terminating itself. Have the original process print 1, the next print 2, the next print 3, and so forth.

Solution: Something like this will work....

```
for(int i = 1; i < N; i++) {
    if (fork() == 0) {
        printf("%d\n", i+1);
        if (i+1 == N) exit(0); // last child exits here
    } else {
        if (i == 1) printf("1\n");
        wait(&status); // wait on child process
        exit(0); // all others exit here
    }
}
```

Problem 2 Discuss any additional hardware and/or software (code and/or data) considerations necessary to permit a multiprocessing OS to

(a) spool processes (jobs). (Discuss only issues peculiar to spooling, as opposed to general multiprocessing issues.)

Solution: Some sort of secondary storage device, like a disk or tape, needs to be physically connected to the computer and it must be possible to transfer from this to main memory in a controllable way. Some sort of “long term” scheduler software is needed too. This would be responsible for monitoring memory usage, detecting when memory becomes available, and arranging for the program code for some job (process) to be loaded from secondary storage into main memory. It must then essentially alert the regular “short term” scheduler (standard part of any multiprocessing system) that this new job is available for CPU scheduling.

(b) implement timesharing to create the illusion of processes executing in parallel.

Solution: Need a clock of course, and in fact, need to arrange for clock-generated hardware interrupt. Also need interrupt handling software for this, which should sense when the current process has exceeded its time slice (quantum) and arrange for this to be put into the ready state and for some other process to be put into the running state.

(c) implement a dynamic priority (timesharing) scheduling scheme that estimates how I/O-bound a process is (based on measurements from a single quantum of CPU usage), and uses this estimate in order to give heavily I/O-bound processes higher priority.

Solution: Measure the fraction f of the process’s allotted quantum that the process actually used before suspending itself, if it in fact did this. When putting this process back into the ready queue, set its priority to something like $1/f$. This way I/O-bound processes tend to get higher priority. (See first complete paragraph on page 86).

Problem 3 Three processes A, B and C start executing (nearly) together in a multiprocessing environment, and require 7, 12 and 16 seconds, respectively, of run time. What will the average turnaround time be if ...

(a) (non-preemptive) Shortest Job First scheduling is used?

Solution:

$$\frac{7 + 19 + 35}{3} = \frac{61}{3} = 20.33$$

(b) (preemptive) Round Robin scheduling is used?

Solution:

$$\frac{21 + 31 + 35}{3} = \frac{87}{3} = 29$$

(c) the (preemptive) scheduling for the CTSS system we discussed is used, where we'll assume that the processes A, B and C start executing in that order, that each initially starts with the highest priority and as such is given an initial quantum of one second. (Hint: recall that later quanta will be larger.)

Solution:

$$\frac{12 + 25 + 34}{3} = \frac{71}{3} = 23.67$$

because *A* runs for first second, *B* runs for next second, *C* runs for next second, *A* runs for next two seconds, *B* runs for next two seconds, *C* runs for next two seconds, *A* runs for next four seconds, *B* runs for next four seconds, *C* runs for next four seconds, *B* runs for next five seconds, and *C* runs for next nine seconds.

Problem 4

(a) Suppose N processes with IDs $0, 1, 2, \dots, N - 1$ need to share some critical region of code, which only one at a time is allowed to execute, and where they are required to repeatedly cycle through this section in the natural order $(0, 1, 2, \dots, N - 1)$. If `turn` is a shared integer variable initialized to zero, and `current_pid` is the ID of the currently executing process, and each executes the following (pseudo-)code, could a race condition occur? Why or why not? Be precise.

```
while( TRUE )
{
    while( turn != current_pid ) /* wait */ ;
    critical_region();
    turn = (turn + 1) % N;
    noncritical_region();
}
```

Solution: No race condition here. Process i must change the value of `turn` from i to $i + 1$ (modulo n) before Process $i + 1$ can escape its busy waiting loop to enter the critical region. It will be the only process in this critical region.

(b) How many semaphores would be required if a semaphore solution to this problem was used instead? Write some pseudo-code for this.

Solution: N semaphores are required. Process i must signal on a semaphore that process $i + 1$ waits on. Let `semaphore` be an array on N semaphores, each initialized to one. Here is pseudo-code which Process i could execute:

```
wait(semaphore[i]);
critical_region();
signal(semaphore[(i+1)%N]);
noncritical_region();
```

Problem 5 Write some pseudo-code to implement

```
sent(rcvr_pid, message_ptr) and receive(sndr_pid, message_ptr)
```

for a buffered blocking message passing system. Let this use a mailbox (a circular buffer) that can hold up to a dozen messages at a time, and which is associated with the receiving process. In fact, let `mailboxes` be an array of such mailboxes that needs to be indexed using the receiver's ID. Let `current_pid` be the ID of the currently executing process. Let `message_ptr->sndr_pid` be used to hold the ID of the sender of the message, and have the `send` method set this value. To protect the array of mailboxes, you should use two arrays of semaphores, so as to protect each mailbox by means of two semaphores. One would be used to protect against over-production (*i.e.* over-stuffing the mailbox) and the other to protect against over-consumption (*i.e.* trying to receive from an empty mailbox).

Solution: This is a little messier than I intended, and as Marc pointed out, we need to just have `receive()` get the next message, regardless of who sent it. I graded accordingly. I will name the two arrays of semaphores `empty` and `full`. `mailboxes` is an array of circular buffers, and so in fact an array of arrays. The problem is (and I forgot to anticipate this) we also need a pair of indices into each circular buffer, one for production and one for consumption. Let `put_index` and `get_index` be arrays of integers used for these purposes. Each `empty` semaphore needs to be initialized to 12, and each `full` semaphore needs to be initialized to 0. Here is my pseudo-code:

```
void sent(rcvr_pid, message_ptr)
{
    message_ptr -> sndr_pid = current_pid;
    wait(empty[rcvr_pid]);
    put_index[rcvr_pid]++;
    put_index[rcvr_pid] %= 12;
    copy *message_ptr to mailbox[rcvr_pid][put_index[rcvr_pid]]
    signal(full[rcvr_pid]);
}

void receive(message_ptr)
{
    wait(full[current_pid]);
    get_index[current_pid]++;
    get_index[current_pid] % 12;
    copy mailboxes[current_pid][get_index[current_pid]] to *message_ptr
    signal(empty[current_pid]);
}
```

Problem 6 See the MINIX code on the next two pages. Recall that `unready(rp)` first checks to see if the process table pointed to by `rp` is at the head of the appropriate ready queue. If so, it is removed, and if this table belongs to currently running process, then another process is selected for execution. However, if it is not at the head of the ready queue, then the rest of the queue is searched, and if it is found, then it is removed from the queue.

(a) If another process is selected for execution, which process will this be. Explain carefully how this selection is made.

Solution: If a task is available (*i.e.* task ready queue is non-empty) then select task at the head of a queue. Otherwise, if a server process is available, then do likewise for this. Otherwise, if a user process is available, then do likewise for this. Otherwise, select the IDLE process.

(b) Carefully rewrite the portion of the code that searches the ready queue, so that this search starts at the tail and moves towards the head, instead. Be efficient here. Caution: don't dereference a null (`NIL_PROC`) pointer.

Solution: Well, as we said, this presumes a doubly linked list, and so the existence of a pointer `p_previousready` in the `proc` structure. In this case, here is some code that will do the job:

```
if (istaskp(rp)) xp = rdy_tail[TASK_Q]
    else if (isuserp(rp)) xp = rdy_tail[USER_Q]
    else xp = rdy_tail[SERVER_Q];
if ( xp == NIL_PROC ) return;
while ( xp != rp )
    if ((xp = xp->p_previousready) == NIL_PROC) return;
if (xp->p_nextready != NIL_PROC)
    xp->p_nextready->p_previousready = xp->p_previousready;
if (xp->p_previousready != NIL_PROC)
    xp->p_previousready->p_nextready = xp->p_nextready;
```

The following is a portion of the Minix 2.0 OS source code from Tannenbaum and Woodhull:

```
/*=====
*                                     pick_proc                                     *
*=====*/
PRIVATE void pick_proc()
{
/* Decide who to run now. A new process is selected by setting 'proc_ptr'.
* When a fresh user (or idle) process is selected, record it in 'bill_ptr',
* so the clock task can tell who to bill for system time.
*/

register struct proc *rp;    /* process to run */

if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
    proc_ptr = rp;
    return;
}
if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
    proc_ptr = rp;
    return;
}
if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
    proc_ptr = rp;
    bill_ptr = rp;
    return;
}
/* No one is ready. Run the idle task. The idle task might be made an
* always-ready user task to avoid this special case.
*/
bill_ptr = proc_ptr = proc_addr(IDLE);
}

/*=====
*                                     ready                                     *
*=====*/
PRIVATE void ready(rp)
register struct proc *rp;    /* this process is now runnable */
{
/* Add 'rp' to the end of one of the queues of runnable processes. Three
* queues are maintained:
* TASK_Q - (highest priority) for runnable tasks
* SERVER_Q - (middle priority) for MM and FS only
* USER_Q - (lowest priority) for user processes
*/

if (istaskp(rp)) {
    if (rdy_head[TASK_Q] != NIL_PROC)
        /* Add to tail of nonempty queue. */
        rdy_tail[TASK_Q]->p_nextready = rp;
    else {
        proc_ptr =          /* run fresh task next */
        rdy_head[TASK_Q] = rp; /* add to empty queue */
    }
    rdy_tail[TASK_Q] = rp;
    rp->p_nextready = NIL_PROC; /* new entry has no successor */
    return;
}
if (!isuserp(rp)) { /* others are similar */
    if (rdy_head[SERVER_Q] != NIL_PROC)
        rdy_tail[SERVER_Q]->p_nextready = rp;
    else
        rdy_head[SERVER_Q] = rp;
    rdy_tail[SERVER_Q] = rp;
    rp->p_nextready = NIL_PROC;
    return;
}
if (rdy_head[USER_Q] == NIL_PROC)
    rdy_tail[USER_Q] = rp;
rp->p_nextready = rdy_head[USER_Q];
rdy_head[USER_Q] = rp;
/*
if (rdy_head[USER_Q] != NIL_PROC)
    rdy_tail[USER_Q]->p_nextready = rp;
else
    rdy_head[USER_Q] = rp;
rdy_tail[USER_Q] = rp;
*/
}
```

```

}
*/

/*=====
*                               unready                               *
*=====*/
PRIVATE void unready(rp)
register struct proc *rp;      /* this process is no longer runnable */
{
/* A process has blocked. */

register struct proc *xp;
register struct proc **qtail; /* TASK_Q, SERVER_Q, or USER_Q rdy_tail */

if (istaskp(rp)) {
/* task stack still ok? */
if (*rp->p_stguard != STACK_GUARD)
panic("stack overrun by task", proc_number(rp));

if ( (xp = rdy_head[TASK_Q]) == NIL_PROC) return;
if (xp == rp) {
/* Remove head of queue */
rdy_head[TASK_Q] = xp->p_nextready;
if (rp == proc_ptr) pick_proc();
return;
}
qtail = &rdy_tail[TASK_Q];
}
else if (!isuserp(rp)) {
if ( (xp = rdy_head[SERVER_Q]) == NIL_PROC) return;
if (xp == rp) {
rdy_head[SERVER_Q] = xp->p_nextready;
pick_proc();
return;
}
qtail = &rdy_tail[SERVER_Q];
} else
{
if ( (xp = rdy_head[USER_Q]) == NIL_PROC) return;
if (xp == rp) {
rdy_head[USER_Q] = xp->p_nextready;
pick_proc();
return;
}
qtail = &rdy_tail[USER_Q];
}

/* Search body of queue. A process can be made unready even if it is
* not running by being sent a signal that kills it.
*/
while (xp->p_nextready != rp)
if ( (xp = xp->p_nextready) == NIL_PROC) return;
xp->p_nextready = xp->p_nextready->p_nextready;
if (*qtail == rp) *qtail = xp;
}

/*=====
*                               sched                               *
*=====*/
PRIVATE void sched()
{
/* The current process has run too long. If another low priority (user)
* process is runnable, put the current process on the end of the user queue,
* possibly promoting another user to head of the queue.
*/

if (rdy_head[USER_Q] == NIL_PROC) return;

/* One or more user processes queued. */
rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
rdy_tail[USER_Q] = rdy_head[USER_Q];
rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
rdy_tail[USER_Q]->p_nextready = NIL_PROC;
pick_proc();
}

```