University of Ottawa
School of Information Technology and Engineering

Course: CSI3131 – Operating Systems
SEMESTER: Summer 2009

Professor: Nour El Kadri
Date: June 26, 2009
Hour: 1:00-3:00
Room: CBY 012

Midterm Exam

Student Name: **Student ID:**

Please answer all questions on the questionnaire.

The exam consists of three (3) parts:

Part 1	**Objective part**	**10 points**
Part 2	**Short Answer/Theory Questions**	**10 points**
Part 3	**Problem Solving**	**23 points**
Total	**3 bonus points**	**40 points**

Part 1: Objective (10 questions, one point each):

Mark True/False questions by filling the box of the right answer. Circle the letter for the multiple choice questions. Your answers to this part should be on the question papers.

1. On a multi-CPU computer, two threads of the same process in a pure user thread environment (many to one model) can be executed on two different CPUs.

   Is this statement TRUE □ or FALSE ■ ?

2. Consider the conditional variable `y` in a monitor. The call to `y.wait()` always blocks the calling thread.

   Is the last statement TRUE ■ or FALSE □ ?

3. List the three main input/output mechanisms. Circle the one(s) that would be most common in operating systems.

   Direct I/O **Interrupts** **DMA**

4. For certain applications, it makes sense to maintain threads across an `exec()` call?

   Is this statement TRUE □ or FALSE ■ ?

5. Which of the following resources are **NOT** shared by threads within the same process:

   a. program counter

   b. open files

   c. the code segment

   d. the data segment

   e. the stack

6. One of the following is true about privileged instructions

   a. When a privileged instruction is executed, the CPU switches to kernel mode.
   b. When a privileged instruction is executed while the CPU is in kernel mode, an exception is raised and the CPU switches back to user mode.
   c. If a privileged instruction is executed while the CPU is in user mode, the user is prompted for the supervisor password.
   d. The only privileged instructions are the input/output instructions.
   e. Executing a privileged instruction while the CPU is in user mode causes exception.

7. One of the following actions does not necessarily occur during context switch from process P:

   a. saving the program counter of P
   b. updating the PCB of P
   c. placing P's PCB in the appropriate queue

d. suspending P
e. choosing the next process to be given the CPU

8. Mark all (there are several of them) resources that are **NOT** shared by threads within the same process:

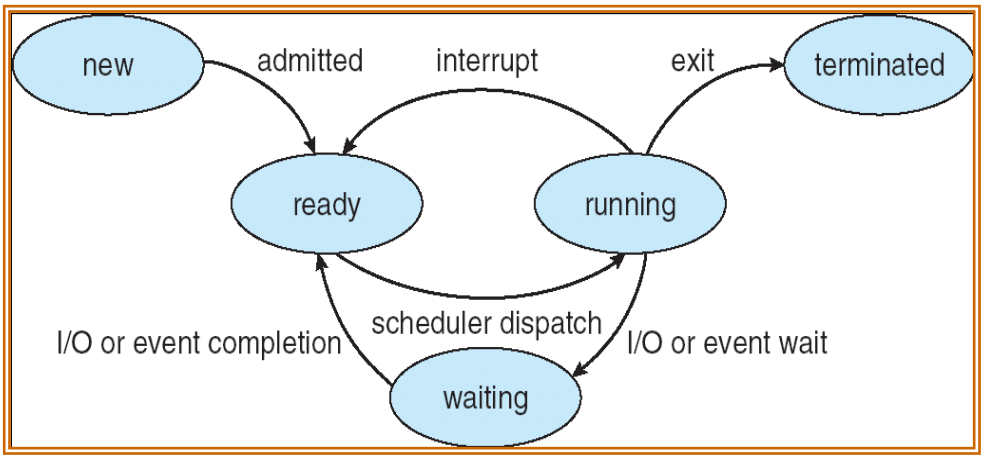
   a. program counter
   b. open files
   c. the code segment
   d. the data segment
   e. the stack

9. One of the following statements is incorrect about pure user threads
   a. Context switches between threads of the same process are done without the involvement of the OS kernel.
   b. On a multi-CPU computer, two threads of the same process can be executed on different CPUs.
   c. When one thread of a process blocks, no other thread of that process will be given the CPU.
   d. There can be many user threads that are mapped to single kernel thread
   e. The threads of the same process can be in different states.

10. One of the following statements is correct about monitors
    a. Mutual exclusion is ensured by the use of conditional variables.
    b. A call to cwait() is always blocking.
    c. With monitors, mutual exclusion must be programmed explicitly.
    d. A call to csignal() always wakes-up another process or thread.
    e. The shared variables to which the monitor guards access are called conditional variables.

Short answer questions (10 points)

1. (5 points) Draw the 5 state process diagram and explain when each of the following transitions occur:
   a. From *new* state to *ready* state
   b. From *ready* state to *wait* state
   c. From *running* state to *ready* state
   d. From *waiting* state to *terminated* state
   e. From *running* state to *wait* state



a- From *new* state to *ready* state

When a process is admitted to the system by the long-term scheduler

b- From *ready* state to *wait* state

This cannot happen by process's own action. If a suspend signal is sent to the process, it will wait until a resume signal is received.

c- From *running* state to *ready* state

Time quantum expired. Preempted by a higher priority process.

d- From *waiting* state to *terminated* state

Again, cannot directly happen by the action of the process itself. Can happen if somebody sends kill signal to the process, or when the process is caught in cascading termination

e- From *running* state to *wait* state

Called a blocking system call (i.e. blocking I/O operation, wait on a semaphore, wait until child terminates, blocking messaging…)

2. (1 point) Give an IPC mechanism that allows communication between two processes running on different computer systems?

Sockets, RPC, RMI

3. (1 point) What are the two CPU operating modes that support the implementation of operating systems?

Kernel mode and user mode.

4. (1 point) What are the two approaches to creating threads in Java?

Extend the class Thread.

Implement the interface Runnable.

5. (2 points total) In Mutual Exclusion solutions:
    a. (1 point) Why use semaphores when we have hardware instructions (test&set, xchng)?

Solutions using hardware instructions involve busy waiting, while with semaphores we can block the process trying to get the CS and use the CPU to do other work. The semaphores are higher level constructs that allow more convenient programming.
Semaphores eliminates (almost) busy-waiting that are used by the hardware instructions.

    b. (1 point) Why use monitors when we have semaphores?

The problem with semaphores is that the synchronization/mutual exclusion code is spread out over all processes accessing the shared variables. It is quite difficult to maintain, as an error in single process might cause disruption everywhere. Monitors are a higher level synchronization tool, which 1) encapsulates the shared variables, so that they can be accesses only through monitor methods. This means that in order to ensure correctness in accessing the variable, it is enough to ensure correctness of the monitor. Moreover, the monitors provide implicit mutual exclusion.
Facilitates synchronization programming by collecting all synchronization logic in a single location (abstract data structure).

Problem Solving (23 points)

Note: Partial solutions will get partial marks.

Problem 1 (8 points): Simulate Priority Round Robin.

Scheduler and system specification:

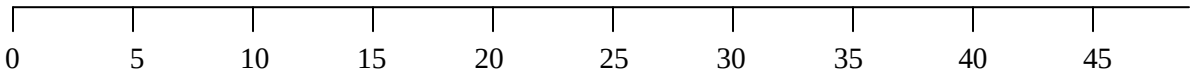
- There are two priority levels: *low* and *high*
- Time quantum for both priority levels is 10 time units
- If a process is preempted because a higher priority process arrives, it remains at the top of its ready queue – it is placed at the end only when its time quantum expires or when its CPU burst is finished.
- Each CPU burst is followed by an I/O operation taking 16 time units
- The I/O operations of different processes overlap and do not interfere, i.e. if P1 starts to wait at time 10 and P2 starts to wait at time 20, P1 will become ready at time 10+16 = 26 and P2 will become ready at time 20+16=36.

Process specification:

- There are four processes P0, P1, P2 and P3
- Processes P0 and P2 arrive at time 0, processes P1 and P3 arrive at time 10
- P0 and P1 are of *high* priority, P2 and P3 are of *low* priority
- The CPU burst time of processes are as follows:
  P0: 2, 3, 2, 3
  P1: 5, 14, 7, 9
  P2: 12, 3, 18, 4
  P3: 15, 25, 32, 2

Your task:

- Simulate the Round Robin scheduler scheduling these 4 processes for the first 40 time steps
- In the following line, mark the times of the context switches and mark which process is executing at a given moment:



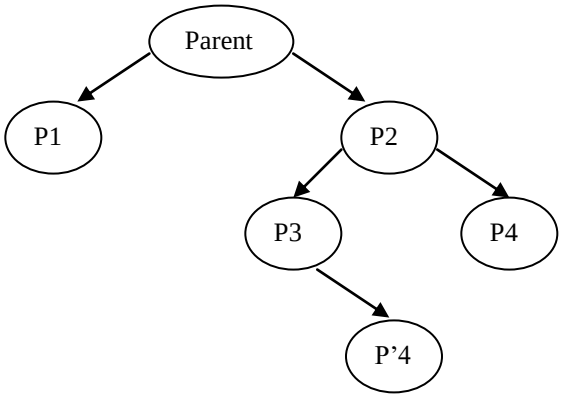
Time	Process	Comment
0	P0	Highest priority scheduled, P2 ready, P1 and P3 not present yet
2	P2	P0 goes to I/O wait, P1 and P3 not present yes
10	P1	P1 and P3 arrive, P2 preempted by P1, P0 still at I/O wait
15	P2	P1 goes to I/O wait, P2 given chance to finish its time slice
17	P3	P2 finished its time slice of 10, the CPU is given to P3 using round robin
18	P0	P0 finishes I/O, becomes ready and preempts P3
21	P3	P0 goes to I/O wait, P3 given chance to finish its time slice
30	P2	P3 finishes its time slice, P0 and P1 still at I/O wait, P2 scheduled
31	P1	P1 finished I/O, becomes ready and preempts P2
41	P0	P1’s time slice expired, P0 is scheduled using round robin

Problem 2 (7 points): Consider the following code snippet (you may assume that fork() never fails)

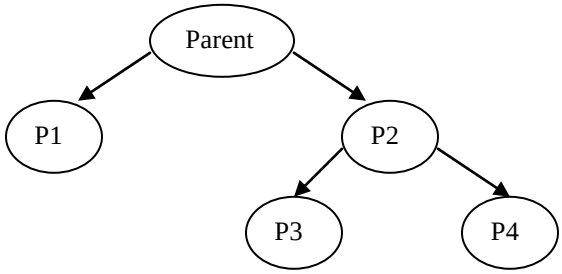
```
if ((p1_pid = fork())== 0)
    printf("I am P1 and I am proud of it.");
} else {
    if ((p2_pid = fork()) == 0) {
        p3_pid == fork();
        printf("I am P3. I like it.");
        p4_pid == fork();
        printf("I am P4. Get used to it.");
    } else
        printf("I am the parent process, obey or die!");

}
```

a) (1 point) How many processes (excluding the parent) will be created?

5, the structure looks as follows:



b) (2 points) Correct the above code so that the following scheme captures the process tree:
(1 point) Modify the code to make sure that P4 is launched only after P1 has terminated.




```
if ((p1_pid = fork())== 0) {
    printf("I am P1 and I am proud of it.");
} else {
```

```
    if ((p2_pid = fork()) == 0) {
        printf("I am P2 and want children!");
        if ((p3_pid = fork()) == 0) {
            printf("I am P3. I like it.");
        } else {
            wait(p1_pid);
            if ((p4_pid = fork()) == 0) {
                printf("I am P4. Get used to it.");
            } else {
                printf("I am P2 and have two children!");
            }
        }
    } else
        printf("I am the parent process, obey or die!");

}
```

c) (2 points) Modify the code so that there is a pipe going from P1 to P3. Make sure that P1 has open only the write end of the pipe, P3 has open only read end of the pipe and nobody else has anything open.

(1 point) Have the standard output of P1 redirected to the write end of that pipe, and the standard input of P3 redirected to the read end of that pipe. The P1's printing "I am P1 and I am proud of it" should already be sent to the pipe.

```
// there should be error checking code everywhere, but
// that is not that important in the exam
pipe(pip13);
if ((p1_pid = fork())== 0) {
    dup2(pip13[1],1);
    close(pip13[0]);
    printf("I am P1 and I am proud of it.");
} else {
    close(pip13[1]);
    if ((p2_pid = fork()) == 0) {
        printf("I am P2 and want children!");
        if ((p3_pid = fork()) == 0) {
            dup2(pip13[0],0);
            printf("I am P3. I like it.");
        } else {
            close(pip13[0]);
            wait(p1_pid);
            if ((p4_pid = fork()) == 0) {
                printf("I am P4. Get used to it.");
            } else {
                printf("I am P2 and have two children!");
```

```
            }
        }
    } else {
        close(pip13[0]);
        close(pip13[1]);
        printf("I am the parent process, obey or die!");
    }

}
```

Problem 3 (8 points): Answer this problem on the question sheet.
Consider the problem of simulating an amusement park ride consisting of a single airplane: The airplane can hold up to TOTAL passengers. The ride operates as follows:

- The passengers arrive at entry to the ride; wait until the airplane is ready for boarding, then board (one by one).
- When the airplane is full, it flies for a period of time, and then lands.
- After the lane landed, the passengers leave.
- When all passengers left, waiting passengers can start boarding (back to the first step).

Passengers and the airplane are represented using different processes. Examine the (pseudocode) solution for this simulation problem provided in the annex (you may detach the page).

This is President Obama's favorite amusement park ride. But since he is President, certain precautions have been taken for security reasons:

- He must ride alone on the airplane with his security guards. No other passenger is allowed on board.
- If the airplane is partially loaded at his arrival, he graciously waits until the next ride. But, no new passenger boards after he arrives. The airplane takes off immediately and upon return, gives the President his ride.

Your task: To accommodate the new President process (he will take only one ride during his short trip to Ottawa):

- Complete the code for the Passenger (next page); note that only the logic for allowing the passenger to start boarding the airplane changes.
- And finally create new code for the President process.

```
/* Declare and initialize any new semaphores and variables here
   – assume that the semaphores/variables defined in the annex are
   already declared */
```

int presidentFlag = FALSE;

Passenger code:

```
while(true) {
   // Hint: Consider using a loop to keep passengers processes
   // blocked on the boarding semaphore after arrival of the
   // president process.
```

```
while(true) {
        noVIPs.wait(); /* wait until there are no VIPs */
        boarding.wait();
        if (vips>0)    /* vips appeared, don't board */
           boarding.signal();
        else {
        /* if there are still no VIPs, tell other passengers
           about it and proceed with  actual boarding */
           noVIPs.signal();
           break;
        }
    }
    /* the rest is the same as before */
    mutex.wait();
    onBoard++; /* board the airplane */
    if (onBoard == CAPACITY) full.signal();
    boarding.signal();
    mutex.signal();
    takeoff.wait();
    /* enjoying the ride */
    leave.wait();
    mutex.wait();
    onBoard--; /* leave the airplane */
    if (onBoard > 0) leave.signal();
    else empty.signal();
    mutex.signal();
    /* enjoy the solid ground, wait until the bowels settle */
```

```
    /* Code does not change after passenger is allowed
       on the airplane */
    onBoard++; /* board the airplane */
    if(onBoard == TOTAL) full.signal();
    else boarding.signal();  // Note that boarding serves as a mutex
                             // for accessing onBoard variable
    timeToFly.wait();
    enjoyRide(); /* enjoying the ride */
    leave.wait();
    onBoard--; /* leave the airplane */
    if (onBoard == 0) empty.signal();
    else leave.signal();  // Note that leave serves as a mutex
                          // for accessing onBoard variable
    /* ride is complete */
    visitPark();  /* Visit the amusement park for a while*/
}
```

President Obama code:

```
presidentRide() // takes only one ride
{


    presidentFlag = TRUE; // hold back passengers
    boarding.wait();
    if(onboard != 0)   // Let one ride go – but immediately
    {
       full.signal();  /* have plane take off */
       boarding.wait();
    }
    onBoard++; /* board the airplane */
    full.signal();
    timeToFly.wait();
    enjoyRide(); /* enjoying the ride */
    leave.wait();
    onBoard--; /* leave the airplane */
    presidentFlag = FALSE;
    empty.signal();
    /* ride is complete – goes back to Washington*/

}
```

Annex A

Part III, Problem 2:

```
int onboard = 0;
Semaphore boarding = new Semaphore(0);
Semaphore full = new Semaphore(0);
Semaphore empty = new Semaphore(0);
Semaphore leave = new Semaphore(0);
Semaphore timeToFly = new Semaphore(0);
```

Airplane code:

```
int i;
while(true) {
    boarding.signal(); /* ready for boarding */
    full.wait();       /* wait until full */
    for(i=0 ; i< onboard ; i++) timeToFly.signal();  /* away we go */
    fly(); /* takeoff and fly, then land */
    leave.signal();
    empty.wait();
    /* all passengers have left */
}
```

Passenger code:

```
while(true) {
    /* arrive to the airport, wait for the airplane */
    boarding.wait();
    onBoard++; /* board the airplane */
    if(onBoard == TOTAL) full.signal();
    else boarding.signal();  // Note that boarding serves as a mutex
                             // for accessing onBoard variable
    timeToFly.wait();
    enjoyRide(); /* enjoying the ride */
    leave.wait();
    onBoard--; /* leave the airplane */
    if (onBoard == 0) empty.signal();
    else leave.signal();  // Note that leave serves as a mutex
                          // for accessing onBoard variable
    /* ride is complete */
    visitPark();  /* Visit the amusement park for a while*/
}
```