University of California, San Diego
Department of Computer Science of Engineering

Midterm Examination 1

CSE120 Operating System Principals
Spring, 2011

9:30-10:50am, May 2nd

Print your name and ID number neatly in the space provided below; print your name at the upper right corner of every page.

Name:
ID:

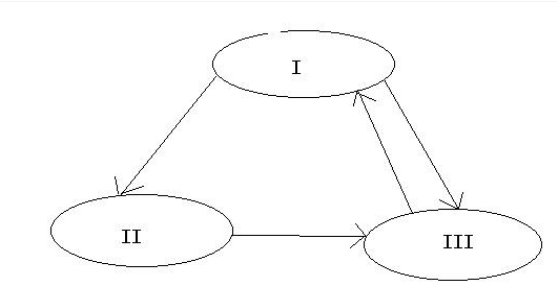
Write your answers CLEARLY. The answer is wrong if the instructor and the TAs cannot read it. This exam has five questions worth a total of 100 points. You have approx. 80 minutes to complete the questions. As with any exam, you should read through the questions first and start with those that you are most comfortable with. If you believe that you cannot answer a question without making some assumptions, state those assumptions in your answer.

Do all parts of all four problems in this booklet and a total of 11 pages. Do your work inside this booklet, using the empty page and backs of pages if needed.

Problem	Score	Grader
1	/30	
2	/28	
3	/16	
4	/14	
5(project-related)	/12	
Total		

1. Multi-Choice Questions (30 Points, 3 points each)

1. Given the following state transition diagram for processes, what are the correct labels for I, II, and III?



A. I, Ready. II, Running. III, Blocked
B. I, Running. II, Ready. III, Blocked
C. I, Running. II, Blocked. III, Ready.
D. I, Blocked. II, Ready. III, Running.
E. I, Ready. II, Blocked. III, Running

Your answer: [C]

2. Which of the following is not a function of an operating system?

A. Generating interrupts
B. Making the computer system convenient to use
C. Manage I/O devices
D. Protecting user programs from one another

Your answer: [A]

3. Which of the following are shared among threads of the same process?

I. Stack
II. Signal handlers
III. Address Space
IV. Program Counter
V. Global variables.

A. II, III, and V
B. III and V
C. I, IV, and V
D. I, II, III, and V
E. I, III, and V

Your answer: [A]

4. Which of the following scheduling algorithms can **suffer** from starvation?

1. FIFO 2. Round-Robin 3. Shortest-Job-First

a) only 1
b) only 2
c) only 3
d) 1 and 2
e) 2 and 3

Your answer: [C]

5. Which of the following may be executed in user mode?

A. Forking a new process
B. Reading a sector from disk
C. Modify the interrupt vector
D. Trap to the kernel
E. Extending a processes address space

Your answer: [D]

6. Place the following memory storage devices in order from slowest access time to fastest access time.

I. Main memory (RAM)
II. Cache
III. Magnetic disk
IV. Registers
V. Magnetic tape

A. II, IV, I, III, V
B. V, III, I, II, IV
C. IV, I, II, III, V
D. IV, II, I, III, V
E. I, II, III, IV, V

Your answer: [B]

7. What is the most common state for a process on a desktop computer?

A. Running
B. Ready
C. Waiting
D. None of the above

Your answer: [B]

8. This code fragment attempts to solve the mutual exclusion problem for two processes. Which of the following problems does it exhibit?

```
int turn;                   /*Whose turn is it?*/
int interested[N];          /*All values initially FALSE*/

void enter_region(int process)
{
    int other;
    other = 1 – process;
    interested[process] = TRUE;
    turn = process;
    while(turn == process &&
          interested[other] == true);
}

void leave_region(int process)
{
    interested[process] = FALSE;
}
```

A. Mutual exclusion
B. Progress
C. Bounded wait
D. None of the above

Your answer: [D]

9. What could be the possible results after the following two threads finish?

```
int X=0;            // a shared variable

Thread 1            Thread 2

X++;                X--;
```

A. 0
B. -1
C. +1
D. -1 or +1
E. 0 or -1
F. 0, -1, or 1

Your answer: [F]

10. What condition for deadlock does the following solution attack to prevent deadlocks?

Order all resources numerically, and at any time a process is only allowed to request for a resource whose numerical value is larger than the values of all resources that are currently held by this process.

A. Circular wait
B. Mutual Exclusion
C. No preemption
D. Hold and wait

Your answer: [A]

2. Synchronization (28 points)

1 (15 points) A process has two kinds of threads; red, and blue. The conditions for any thread to be able to enter its critical section are the following.

1.1 (12 points)

A) Multiple red threads may access the shared variables at the same time, during which time no blue threads may access the shared variables.

B) Multiple blue threads may access the shared variables at the same time, during which time no red threads may access the shared variables.

Write the synchronization code for each color thread. The variable and function declarations are provided for you. You can use **only** the following synchronization primitives. Define additional variables if you find it necessary. You may write in pseudocode or languages other than C as long as the syntax can be easily understood.

Primitives

- **lock(Mutex m)**
  Semantics: acquires the lock on m; other threads invoking lock on m before it is unlocked are appended to the waiting queue on m and put to sleep
- **unlock(Mutex m)**
  Semantics: releases the lock on m; removes one thread(if any) from the waiting queue on m and awakens it
- **wait(Condition cond, Mutex m)**
  Semantics: appends the thread to the waiting queue on cond; automatically unlocks m; the thread is then put to sleep
- **signal(Condition cond)**
  Semantics: removes one of the threads(if any) from the waiting queue on cond and awakens it
- **broadcast(Condition cond)**
  Semantics**:** removes all threads(if any) from the waiting queue on cond and awakens them

```
/* some variables already declared for you  */
Mutex m;
Condition c;
int numRed=0, numBlue=0;

/* You are to implement the following functions. */
void red_enter() {      //this is called by the red thread before entering the critical section
    lock(m);
    while (numBlue > 0) {
        wait(c, m);
        lock(m);
    }
    numRed++;
    unlock(m);
}
void red_leave() {    //this is called by the red thread after finishing the critical section
    lock(m);
    if (--numRed == 0)
        broadcast(c);
    unlock(m);
}
void blue_enter() {    //this is called by the blue thread before entering the critical section
    lock(m);
    while (numRed > 0) {
        wait(c, m);
        lock(m);
    }
    numBlue++;
    unlock(m);
}
void blue_leave() {  //this is called by the blue thread after finishing the critical section
    lock(m);
    if (--numBlue == 0)
        broadcast(c);
    unlock(m);
}
```

-1 pt for each incorrect lock/unlock call.

-8 pts if doesn't use condition variables correctly

1.2 (6 points) Does your implementation have a starvation problem? Please explain how a starvation could occur.

Yes. Once one color of thread acquires the lock, a steady stream of that color thread can keep the lock. This starves threads of the other color.

2 (10 points) Show how a lock, and acquire( ) and release( ) functions can be implemented using atomic SWAP instruction. The following is the definition of swap instruction:

```
void Swap (char* x,* y) \\ The following is done atomically as one hardware instruction
{    char temp = *x;
     *x = *y;
     *y = temp
}
struct lock {
    char held = "0";

}

void acquire (struct lock * lck) {
    char localVal = "1";
    do {
        swap(&lck->held, &localVal);
    while (localVal == "1");

}
void release (struct lock * lck) {
    char localVal = "0";
    swap(&lck->held, &localVal);

}
```

-1 pt for a minor mistake.

-3 pts for each section that is incorrect

3. Process Scheduling (16 Points)

Suppose we have 4 processes arrive for execution. The process burst time (duration), as well as the arrival time and priority of each process is indicated in the following table. Time is in milliseconds (ms).

Process	Arrival Time	Burst Time (duration)	Priority
P_1	6	6	4
P_2	2	10	3
P_3	0	2	2
P_4	1	7	4

Table 1: The processes with their arrival time, process behavior and priority

Fill out the three Gantt charts below illustrating the scheduled execution of these processes using

a) Priority scheduling (6 points)
b) Round Robin (quantum = 2ms) (10 points)

Note: CPU cannot stay idle if there are still unfinished processes.

Gantt Chart for Priority Scheduling

0-2 : P_3
2-12 : P_2
12-19 : P_4
19-25 : P_1

-2 pts if switched P_1 and P_4
-4 pts if didn't get the seq right at all

Gantt Chart for Round Robin Scheduling with time slice 2ms

Incorrect old solution:

	P_3	P_4	P_2	P_1	P_4	P_2	P_1	P_4	P_2	P_1	P_4	P_2
Time: 0	2	4	6	8	10	12	14	16	18	20	21	25

Correct solution:

	P_3	P_4	P_2	P_4	P_1	P_2	P_4	P_1	P_2	P_4	P_1	P_2
Time: 0	2	4	6	8	10	12	14	16	18	19	21	25

-5 pts if missed the seq somewhere.
-10 if didn't repeat in round robin fashion

4. Deadlock (14 points)

A restaurant would like to serve four dinner parties, P1 through P4. The restaurant has a total of 8 plates and 12 bowls. Assume that each group of diners will stop eating and wait for the waiter to bring a requested item (plate or bowl) to the table when it is required. Assume that the diners don't mind waiting. The maximum request and current allocation tables are shown as follows:

Maximum Request	Plates	Bowls
P1	7	7
P2	6	10
P3	1	2
P4	2	4

Current Allocation	Plates	Bowls
P1	2	3
P2	3	5
P3	0	1
P4	1	2

(a) (4 Points) Determine the Need Matrix for plates and bowls.

Need	Plates	Bowls
P1	5	4
P2	3	5
P3	1	1
P4	1	2

(b) (10 Points) Will the restaurant be able to feed all four parties successfully? Explain your reason.

No. Can serve $\mathbf{P_3}$ and $\mathbf{P_4}$ but then will not have enough resources to serve $\mathbf{P_1}$ or $\mathbf{P_2}$.

5. Project-Related Questions (12 points)

1. (6 points) Nachos provides an implementation of condition variables in Condition.java that uses semaphores. (Do not confuse this implementation with your own in Condition2.java.) In the `sleep()` method in Condition.java, the following lines of code appear:

```
Semaphore waiter = new Semaphore(0);
waitQueue.add(waiter);
conditionLock.release();
waiter.P();
conditionLock.acquire();
```

Recalling how the Condition.java implementation works, which of the following is true?

A. Semaphore `waiter` is used to sleep the current thread.
B. Semaphore `waiter` is incremented when the current thread calls `waiter.P()`.
C. The current thread must be the holder of `conditionLock` before calling `sleep()`.
D. All are true.
E. Both B & C.
F. Both A & C.

Your answer: [F]

2. (6 points) In Project 1, you were asked to implement Alarm.waitUntil(). A solution is presented below.

```
public void waitUntil(long x) {
    long wakeTime = Machine.timer().getTime() + x;
    while (wakeTime > Machine.timer().getTime())
        KThread.yield();
}
```

Based on the solution, which of the following is true?

A. It only works for 1 thread.
B. It's a form of busy waiting.
C. KThread.yield() might cause the thread to block indefinitely.
D. All are true.
E. B & C
F. A & B

Your answer: [B]

This is a blank page for answer overflow**. If you need to use this page, please write the problem No. of your solution clearly.**