Name, SURNAME and ID ⇒ **KEY**

Middle East Technical University
Department of Computer Engineering

CENG 334

Section 1 and 2
Spring '2009-2010

Midterm

- **Duration: 120** minutes.
- **Exam:**
  - This is a **closed book**, **closed notes** exam. No attempts of cheating will be tolerated. In case such attempts are observed, the students who took part in the act will be prosecuted. The legal code states that students who are found guilty of cheating shall be expelled from the university for **a minimum of one semester**!
- **About the exam questions:**
  - The points assigned for each question are shown in parenthesis next to the question.
  - Whereever available, use the boxes to write down your answers.
- **This booklet consists of *8* pages including this page. Check that you have them all!**
- **GOOD LUCK !**

Question 1	
Question 2	
Question 3	
Question 4	
Total ⇒	

1 (20 pts)

Answer the following questions as (T)rue or (F)alse in the table below. Each correct answer will be awarded with 2 points and each wrong answer will be punished with -2 points, hence do not make wild guesses.

1	2	3	4	5	6	7	8	9	10
T	F	F	F	F	T	F	T	F	T

(1) Context switch between the threads of a process can be handled at user-level.

(2) The fork system call takes much longer to execute when called from a large program, because it must copy the program's code segments.

(3) Round-robin scheduling policy is non-preemptive.

(4) Race conditions may happen when multiple threads (or processes) tries to read (but not write) a shared variable.

(5) In a Mesa-style monitor, the thread that signal()'s to wake up other sleeping threads is immediately blocked.

(6) DMA is used to offload from the operating system (processor) the overhead of copying data between I/O devices and main memory.

(7) In a system where some resources have multiple instances (such as two CPU's) the following is true: Cycle in resource allocation graph $\Longleftrightarrow$ Deadlock in the system.

(8) Peterson's algorithm uses spinlocking for synchronization.

(9) Semaphore's are more powerful than mutexes (meaning that some problems can be only solved using semaphores, but not using mutexes).

(10) A group of threads or processes can never deadlock if they acquire any needed resources according to some fixed ordering they all agree on.

2 (25 pts)

Three processes are at the ready queue of the scheduler in the order **A**, **B** and E at time 0. If you use the FCFS policy the scheduling executed is as follows. In the timing diagram shown below for 20 time units, A, B, an E represents the CPU use of these processes whereas b and e denote the I/O's of processes B and E respectively (A has no I/O). The vertical bars indicate transitions of a process from running to the I/O wait or ready queue.

In the diagram below fill in the scheduling executed by Round robin (RR) with a timeslice of 1 unit, Shortest-Job-First (SJF), and Shortest-Remaining-Time-First (SRTF). For SRTF, assume that the scheduler has complete knowledge on the CPU and I/O requirements of the processes. If at a given time slice the CPU is being used by, say A, whereas the I/O requests of B and E are active, then fill in that time slice with Abe **vertically**. Also assume that the I/O device can concurrently handle multiple I/O requests. In SRTF, if the remaining CPU burst of the running process equals to the remaining CPU burst of another process in the ready list, no context switch should occur.

	0										1									
Time	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
FCFS	A		B			b		B			b		B			b		E	e	
RR	A	B	E	A e	B	B	b	b	B	B	B	b	b	B	B	B	b	b		
SJF	E	A e	A	B	B	B	b	b	B	B	B	b	b	B	B	B	b	b		
SRTF	E	A e	A	B	B	B	b	b	B	B	B	b	b	B	B	B	b	b		

RR (9 pts), SJF (8 pts), SRTF (8 pts)
First 6 columns of each policy is 1 point each.
The rest of the row takes the remaining points.

3 (30 pts)

A particular river crossing is shared by both Linux Hackers and Microsoft employees. A boat is used to cross the river, but it only seats four people, and must always carry a full load. In order to guarantee the safety of the hackers, you cannot put three employees and one hacker in the same boat (because the employees would gang up and convert the hacker). Similarly, you cannot put three hackers in the same boat as an employee (because the hackers would gang up and convert the employee). All other combinations are safe.

Two procedures are needed: `HackerArrives()` and `EmployeeArrives()`, called by a hacker or employee when he/she arrives at the river bank. The procedures arrange the arriving hackers and employees into safe boatloads; once the boat is full, one thread calls `Rowboat()` and only after the call to `Rowboat()`, the four threads representing the people in the boat can return (i.e. exit). Note that the `Rowboat()` function needs to be executed by only ONE thread, and all the other threads in the boat do not need to make such a call and can just exit after that.

Any order is acceptable and there should be no busy-waiting and no undue waiting - hackers and employees should not wait if there are enough of them for a safe boatload. Your code should be clearly commented, in particular, you should comment each semaphore or condition variable operation to specify how correctness properties are preserved.

- Solve this concurrency problem using semaphores. You can declare and initialize semaphores as `sem = semaphore(1);` and use them by calling two methods as `sem.up();` or `sem.down();`. Write the code for `HackerArrives()` function in the skeleton shown below. Note that although a number of declaration are provided for you, you are free to declare and use more int's or semaphores as necessary.

  The solution for `EmployeeArrives()` would be symmetric, and you don't have to write that down.

```
hackers = semaphore(0); // Start with zero hackers
employees = semaphore(0); // Start with zero employees
int hackerCount = 0; // Number of waiting hackers
int employeeCount = 0; // Number of waiting employees
Semaphore Mutex = 1; // Semaphore for critical region (checking counts)
Semaphore Rowing =0; // Semaphore to prevent riders from returning from
// Hacker/Employee Arrival call until Rowboat call

void HackerArrives(){
Mutex.down(); // Acquire lock for rider variables
if (HackerCount == 3) { // Three other waiting hackers
    HackerCount -= 3; // Decrement count of waiters
    Mutex.up(); // Release lock
    Hackers.up(); // Wake up three other waiting hackers
    Hackers.up();
    Hackers.up();
} else if ((HackerCount >= 1) && (EmployeeCount >=2)) {
    HackerCount -= 1 // Decrement count of waiters;
    EmployeeCount -= 2;
    Mutex.up(); // Release lock
    Hackers.up();
    Employees.up();
    Employees.up();
} else {
    HackerCount += 1; // New waiting hacker
    Mutex.up(); // Release lock
    Hackers.down(); // Go to sleep until other riders arrive to
                                 // fill boat
    Rowing.down(); // Wait for Rowboat, once we get in the boat
    return;
}
// Only the rider that fills the boat (didn't sleep)
// makes it to this point
RowBoat();
Rowing.up(); // Wake up waiting boat occupants
Rowing.up();
Rowing.up();
}

First three hackers (H1, H2, H3 in marking) end up waiting: 5 pts
2 hackers + 2 employees go together: 5 pts
4 hackers : 5 pts
no mutex use: -3 pts
no/wrong RowBoat: -3 pts
wrong syntax: -3  pts
```

- Solve the same problem using Mesa-type monitors and condition variables. Use the following skeleton to write your code. Note that although a number of declaration are provided for you, you are free to declare and use more int's or condition variables as necessary. Assume that the functions `wait(cv)`, `notify(cv)`, and `notifyAll(cv)`, are available where `cv` is a declared condition variable.

```
monitor HackerEmployee{
int hackerCount = 0; // Number of waiting hackers
int employeeCount = 0; // Number of waiting employees
condition Hacker; // Used to wait for enough people to cross river
condition Employee; // Same as Hacker
void HackerArrives(){
    if (hackerCount == 3) { // three waiting hackers. Lets go!
        notify(Hacker); // Wake three hackers (any three)
        notify(Hacker);
        notify(Hacker);
        hackerCount -= 3; // Decrement state vars (this must be done here,
        Rowboat(); // Cross the river
    } else if ((hackerCount >= 1) && (employeeCount >= 2) { // 1 other hac
        notify(Hacker); // Wake up one hacker, two employees
        notify(Employee);
        notify(Employee);
        hackerCount--; // Decrement state vars
        employeeCount -= 2;
        Rowboat();
    } else {
        hackerCount++; // Wait for more Hackers to arrive
        wait(Hacker(Hacker); // No need to check state vars
    }
}
void EmployeeArrives(){
    // The solution would be symmetric. No need to code here..
}
}

First three hackers (H1, H2, H3 in marking) end up waiting: 5 pts
2 hackers + 2 employees go together: 5 pts
4 hackers : 5 pts
use of (needless) mutex use: -3 pts
no/wrong RowBoat: -3 pts
wrong syntax: -3  pts
notifyAll(employees): -3 pts
```

$\boxed{4}$ (25 pts)

Consider the following snapshot of a system with five processes $(p1, ...p5)$ and four resources $(r1, ...r4)$. There are no current outstanding queued unsatisfied requests.

Currently available resources

r1	r2	r3	r4
2	1	0	0

	Allocation				Maximum Need				May still Need			
Process	r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4
p1	0	0	1	2	0	0	1	2	0	0	0	0
p2	2	0	0	0	2	7	5	0	0	7	5	0
p3	0	0	3	4	6	6	5	6	6	6	2	2
p4	2	3	5	4	4	3	5	6	2	0	0	2
p5	0	3	3	2	0	6	5	2	0	3	2	0

- (5 pts) Compute what each process still might request and fill in the *May still Need* columns.
- (10 pts) Is this system currently deadlocked, or will any process become deadlocked? Why or why not? If not, give an execution order.
  Answer: Not deadlocked and will not become deadlocked. the process finishing order: p1, p4, p5, p2, p3.
- (10 pts) If a request from p3 arrives for (0, 1, 0, 0), can that request be safely granted immediately? In what state (deadlocked, safe, unsafe) would immediately granting the whole request leave the system? Which processes, if any, are or may become deadlocked if this whole request is granted immediately?
  Change available to (2, 0, 0, 0) and p3's row of "may still needs" to (6, 5, 2, 2). Now p1, p4, and p5 can finish, but with available now (4, 6, 9, 8) neither p2 nor p3's "may still need" can be satisfied. So, it is not safe to grant p3's request. Correct answer NO. State is unsafe as the system may or may not deadlock. Processes p2 and p3 may deadlock .

2010 Darwin Awards

Named in honor of Charles Darwin, the father of evolution, the Darwin Awards commemorate those who improve our gene pool by removing themselves from it.

In the late fall and early winter months, snow-covered mountains become infested with hunters. One ambitious pair climbed high up a mountain in search of their quarry. The trail crossed a small glacier that had crusted over. The lead hunter had to stomp a foot-hold in the snow, one step at a time, in order to cross the glacier.

Somewhere near the middle of the glacier, his next stomp hit not snow but a rock. The lead hunter lost his footing and fell. Down the crusty glacier he zipped, off the edge and out of sight.

Unable to help, his companion watched him slide away. After a while, he shouted out, ”Are you OK?”

”Yes!” came the answer.

Reasoning that it was a quick way off the glacier, the second hunter plopped down and accelerated down the ice, following his friend. There, just over the edge of the glacier, was his friend...holding onto the top of a tree that barely protruded from the snow.

There were no other treetops nearby, nothing to grab, nothing but a hundred-foot drop onto the rocks below. As the second hunter shot past the first, he uttered his final epitaph: a single word, which we may not utter lest our mothers soap our mouths.