EECE.4810/EECE.5730: Operating Systems

Spring 2018

Exam 2
March 28, 2018

Name: _____

Section: **EECE.4810 (undergraduate)** **EECE.5730 (graduate)**

For this exam, you may use one 8.5" x 11" double-sided page of notes. All electronic devices (e.g., cell phones, calculators, laptops) are prohibited. If you have a cell phone, please turn it off prior to the start of the exam to avoid distracting other students.

The exam contains 4 questions for a total of 100 points. Please answer the questions in the spaces provided. If you need additional space, use the back of the page on which the question is written and clearly indicate that you have done so.

Please note that students enrolled in EECE.5730 must complete two extra problems, which are worth a total of 15 points:

- Question 3c, on page 9
- Question 4c, on page 11

You will have one hour and fifteen minutes to complete this exam.

Q1: Monitors & semaphores	/ 32
Q2: Deadlock	/ 20
Q3: Scheduling	/ 28 + 8
Q4: Memory management	/ 20 + 7
TOTAL SCORE	/ 100 + 15

1. (32 points) ***Monitors and semaphores***

a. (16 points) A semaphore and its functions can be implemented as a monitor. Remember that a semaphore supports two functions, `down()` and `up()`, as described below:

- `down()`: wait for semaphore value to become positive, then atomically decrement by 1
- `up()`: add 1 to semaphore value, then wake up thread waiting for value to be positive.

Write `down()` and `up()` for the monitor-based semaphore defined below, which assumes the use of Pthreads. You may write your solution using pseudocode if you don't remember the exact syntax for the Pthread functions.

```
typedef struct {
   int val;                // Semaphore value
   pthread_mutex_t lock;   // Lock for semaphore value
   pthread_cond_t posVal; // Cond. var. to wait for val to be > 0
} semaphoreMonitor;

void down(semaphoreMonitor *M) {









}

void up(semaphoreMonitor *M) {









}
```

1 (continued)
Parts b and c of Question 1 involve the "sleeping barber" problem, a classic synchronization problem using two thread types, barber and customer, and a waiting room to hold customers:

- When the barber finishes with a current customer, he checks the waiting room. If a customer is waiting, he removes the customer from the waiting room and indicates he's available to cut hair. Otherwise, the barber sleeps until a new customer arrives.
- When a customer arrives, he checks the barber's status. If the barber is sleeping, the customer wakes the barber up. Otherwise, the customer enters the waiting room.
    - If a seat is available, the customer takes it; otherwise, he leaves (without waiting).

There may be multiple barbers and customers, with up to N customers waiting in the waiting room. The solution requires three semaphores and a shared variable to track free seats in the waiting room, initialized as shown below. Assume "semaphore" is a valid type that is not related to your solution to Question 1a.

```
semaphore barbers = 0;   // # free barbers
semaphore customers = 0; // # waiting customers
semaphore mutex = 1;     // Used if mutual exclusion needed
int freeSeats = N;       // # free seats in waiting room
```

b. (8 points) Use the space below to complete the code for the barber threads. You may write pseudo-code but should use appropriate semaphore functions when necessary.

```
void barber() {
  while (true) {    // Repeat barber process indefinitely









    cut_hair();     // Actual "work" of performing haircut

  }
}
```

1 (continued)

c. (8 points) Use the space below to complete the code for the customer threads. Recall:

- When a customer arrives, he checks the barber's status. If the barber is sleeping, the customer wakes the barber up. Otherwise, the customer enters the waiting room.
    - If a seat is available, the customer takes it; otherwise, he leaves (without waiting).

There may be multiple barbers and customers, with up to N customers waiting in the waiting room. The solution requires three semaphores and a shared variable to track free seats in the waiting room, initialized as shown below. Assume "semaphore" is a valid type that is not related to your solution to Question 1a.

```
semaphore barbers = 0;    // # free barbers
semaphore customers = 0; // # waiting customers
semaphore mutex = 1;      // Used if mutual exclusion needed
int freeSeats = N;        // # free seats in waiting room
```

You may write pseudo-code but should use appropriate semaphore functions when necessary.

```
void customer() {



   if (_____________________________) { // Check if seat available









      get_haircut();       // Actual "work" of getting haircut

   }

   else {



   }
}
```

2. (20 points) ***Deadlock***

a. (12 points) Your system is currently running 3 processes: A, B, and C. Each process requests two types of resources: files and network connections. Assume the system can handle up to 5 open files and 5 network connections (abbreviated "conn.") in total, across all processes.

Each process makes the resource requests shown in the table below, which lists both total requested resources and a list of individual requests. Each request consists of a unique ID (A1 or B2, for example) and the resources requested (files, connections, or both).

Each process only frees resources when it ends, which occurs after all its requests are complete.

Use the Banker's Algorithm to find an order in which all resource requests can be satisfied and list that order in the space provided. You must show the total amount of each resource type used at each step of your solution. Requests from the same process must be in order (for example, A1 must be satisfied before A2), but requests from different processes may be in any order (A1 may be satisfied before or after B1).

	Process A	**Process B**	**Process C**
Total requested resources	4 files 2 connections	2 files 4 connections	3 files 3 connections
Individual resource requests	A1: 3 files A2: 2 conn. A3: 1 file	B1: 3 conn. B2: 2 files B3: 1 conn.	C1: 1 conn. C2: 1 file C3: 2 files, 2 conn

2 (continued)

b. (8 points) You are running a program with two concurrent threads, each of which start execution by obtaining two locks, L1 and L2, as shown below:

Thread 1	Thread 2
`lock(&L1);`	`lock(&L2);`
`lock(&L2);`	`lock(&L1);`
…	…

Are the two threads guaranteed to deadlock? If so, explain why; if not, show an example of how they could execute without deadlocking.

3. (28 + **8** points) ***Scheduling***

a. (20 points) Consider the following set of processes, with the length of the CPU burst time given in milliseconds. Processes may begin executing 1 ms after they arrive (i.e., a process arriving at time 5 could start executing at time 6). Any process arriving at time 0 is in the ready queue when the scheduler makes a decision about the first process to run.

Process	Burst	Priority	Arrival time
P1	5	4	0
P2	7	1	5
P3	6	3	5
P4	2	2	8
P5	4	2	10

Determine the turnaround time for each process using each of the following scheduling algorithms: (i) round-robin (RR) with time quantum = 1 ms, (ii) shortest time to completion first (STCF), and (iii) a non-preemptive priority scheme in which lower numbers indicate higher priority. You do not have to calculate the average turnaround time for each algorithm.

To break ties between processes (same burst time/priority), use first-come, first-serve ordering.

Use the space on this page and the following page to solve this problem.

3 (continued)
Additional space to solve Question 3a:

3 (continued)

b. (8 points) What is the main difference between preemptive and non-preemptive scheduling algorithms? Give one benefit of each of these scheduling algorithm types.

c. (8 points, ***EECE.5730 only***) Which scheduling algorithm would you choose in each of the following situations? Briefly explain your answer in each case.

i. Every process has a burst time of 4 ms.

ii. The first process in the ready queue has a burst time of 100 ms. New processes arrive every 10 ms, and each of these new processes has a burst time of 2 ms.

4. (20 + **7** points) ***Memory management***

a. (14 points) You are given the following lists of holes and address space requests:

- Holes: 481 KB, 573 KB, 300 KB, 100 KB, 900 KB
- Address spaces: 120 KB, 360 KB, 500 KB, 95 KB, 400 KB

Show into what holes the address spaces are allocated and how the hole list changes over time using (i) first-fit allocation and (ii) best-fit allocation.

4 (continued)

b. (6 points) Explain how a virtual address is translated to a physical address using base and bounds address translation. Provide an example translation to support your answer.

c. (7 points, ***EECE.5730 only***) One potential downside of best-fit memory allocation is the time required to identify the best fit for each address space request. How could you improve the performance of best-fit allocation?