EECE.4810/EECE.5730: Operating Systems

Spring 2018

Exam 2 Solution

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;
```

Solution written in bold; didn't write pthread_ function names, but "lock()" should be "pthread_mutex_lock()", etc.

```
void down(semaphoreMonitor *M) {
   lock(&M->lock);              // Lock access to val
   while (M->val == 0)          // Wait for value to be > 0
      wait(&M->lock, &M->posVal);
   M->val--;                    // Decrement semaphore value
   unlock(&M->lock);            // Release lock
}

void up(semaphoreMonitor *M) {
   lock(&M->lock);          // Lock access to val
   M->val++;                // Increment value
   signal(&M->posVal);      // Wake up thread waiting for positive
                            //  value (broadcast would work too)
   unlock(&M->lock);        // Release lock
}
```

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 <u>not related to your solution to Question 1a.</u>

```
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.

Solution in bold.

```
void barber() {
   while (true) {    // Repeat barber process indefinitely
      down(customers);    // Try to acquire customer, sleeping
                          //  if there are none
      down(mutex);        // Must lock access to freeSeats
      freeSeats++;        // Indicate seat is free
      up(barbers);        // Let customer know barber's available
      up(mutex);          // Release "lock"

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

1 (continued)

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

Solution in bold.

```
void customer() {
   down(mutex);            // Lock access to freeSeats

   if (freeSeats > 0) {                // Check if seat available
      freeSeats--;         // Take seat
      up(customers);       // Indicate customer is waiting
      up(mutex);           // Unlock "lock"
      down(barbers);       // Try to "acquire" barber

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

   else {
      up(mutex);           // Must release "lock" before leaving
   }
}
```

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*

Solution: This problem turned out to be much simpler than I planned, for the simple reason that you don't have to interleave anything—having each process run start to finish (for example, A1-A3, B1-B3, then C1-C3) would satisfy the problem requirements.

If you do decide to interleave requests, there are far more ways to do that than I can possibly count for this solution. So, one potential solution would be:

1) A1 (total: 3 files, 0 connections)
2) C1 (total: 3 files, 1 connection)
3) A2 (total: 3 files, 3 connections)
4) C2 (total: 4 files, 3 connections)
5) A3 (total: 5 files, 3 connections)
   → A completes (total: 1 file, 1 connection)
6) C3 (total: 3 files, 3 connections)
   → C completes (no files or connections used)
7) B1 (total: 0 files, 3 connections)
8) B2 (total: 2 files, 3 connections)
9) B3 (total: 3 files, 3 connections)
   → B completes (no files or connections used)

Note: If you're studying this problem for a future exam, know that I'll probably write similar problems exams with some initial resource allocation (for example, Process A has 1 file and Process B has 1 connection) that will force you to interleave requests to complete everything.

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.

Solution: Deadlock isn't guaranteed—if either thread obtains both locks before the other thread obtains one, then both threads will be able to run to completion. Deadlock would only occur if each thread obtained just one lock.

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.

Solution: This problem is very similar to the example problem we did in Lecture 13. The solution below shows three columns: start time (St), end time (End), and turnaround time (TT). As with the example in class, a process with a burst time of 1 starts and ends in the same cycle. You may have used slightly different notation.

	(i) RR			(ii) STCF			(iii) Priority		
Proc	**St**	**End**	**TT**	**St**	**End**	**TT**	**St**	**End**	**TT**
P1	1	5	**5**	1	5	**5**	1	5	**5**
P2	6	24	**19**	18	24	**19**	6	12	**7**
P3	7	23	**18**	6	13	**8**	19	24	**19**
P4	10	14	**6**	9	10	**2**	13	14	**6**
P5	11	21	**11**	14	17	**7**	15	18	**8**

Detailed schedules for the two preemptive schemes (round robin and STCF) are shown on the next page.

3a (continued)
Detailed schedule for round robin, with the final cycle for each process shown in bold:

Time	1	2	3	4	**5**	6	7	8	9	10	11	12	13	**14**	15	16
Process	P1	P1	P1	P1	**P1**	P2	P3	P2	P3	P4	P5	P2	P3	**P4**	P5	P2

Time	17	18	19	20	**21**	22	**23**	**24**
Process	P3	P5	P2	P3	**P5**	P2	**P3**	**P2**

When a new process arrives, it is added to the end of the scheduling queue. The above solution uses what I believe is the simplest method—order the queue based on arrival order, so the processes are simply added from P1 to P5—as we did with the in-class example.

Detailed schedule for STCF, with the time in that process alone in parentheses (for example, "P1 (3-5)" would indicate a 3 ms block in which P1 ran for its 3rd, 4th, and 5th milliseconds):

Time	1 5	6 8	9 10	11 13	14 17	18 24
Process (time)	P1 (1-5)	P3 (1-3)	P4 (1-2)	P3 (4-6)	P5 (1-4)	P2 (1-7)

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.

Solution: Preemptive scheduling algorithms (like STCF or round robin) can force the currently running process to give up the processor to allow another process to run, while non-preemptive schedulers allow each process to run to the end of its scheduled CPU burst.

A preemptive scheduler can improve performance based on certain metrics—for example, STCF minimizes turnaround time by allowing the shortest possible jobs to run as soon as possible, while round robin is the fairest of all scheduling metrics. Non-preemptive schedulers minimize delays due to context switching and are typically simpler to implement.

*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.

Solution: Since every process takes the same amount of time, there's no benefit to having a complex scheduling algorithm, so first-come, first-served scheduling makes the most sense.

Another valid answer, if you assume these processes may have different priorities, would be to use priority scheduling.

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.

Solution: To avoid needless delays for the short, frequently-arriving processes, use a pre-emptive scheme: either shortest time to completion first scheduling or round robin.

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.

Solution: Changes to the hole list are shown in bold at each step. All hole sizes are in KB.

(i) First-fit allocation:

- 120 KB → allocated to 481 KB hole
  - Updated hole list: **361**, 573, 300, 100, 900
- 360 KB → allocated to 361 KB hole
  - Updated hole list: **1**, 573, 300, 100, 900
- 500 KB → allocated to 573 KB hole
  - Updated hole list: 1, **73**, 300, 100, 900
- 95 KB → allocated to 300 KB hole
  - Updated hole list: 1, 73, **205**, 100, 900
- 400 KB → allocated to 900 KB hole
  - Updated hole list: 1, 73, 205, 100, **500**

(ii) Best-fit allocation:

- 120 KB → allocated to 300 KB hole
  - Updated hole list: 481, 573, **180**, 100, 900
- 360 KB → allocated to 481 KB hole
  - Updated hole list: **121**, 573, 180, 100, 900
- 500 KB → allocated to 573 KB hole
  - Updated hole list: 121, **73**, 180, 100, 900
- 95 KB → allocated to 100 KB hole
  - Updated hole list: 121, 73, 180, **5**, 900
- 400 KB → allocated to 900 KB hole
  - Updated hole list: 121, 73, 180, 5, **500**

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.

Solution: Address translation using base and bounds requires two steps (usually in parallel):

- Make sure the virtual address is less than the bound for the address space
- Add the base address to the virtual address to get the physical address

For example, given a base address of 1000, a bound of 500, and a virtual address of 250, since 250 < 500, the translation is valid, and the physical address would be 1000 + 250 = 1250.

*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?*

Solution: One potential solution is to sort the hole list by size so the best fit can be found without searching the entire list—keeping an ordered hole list would reduce the time to search the list to that of a first-fit allocation. While reordering the list would require some time, that time could likely be overlapped with other parts of the allocation process.