EECE.4810/EECE.5730: Operating Systems

Spring 2019

Exam 2 Solution

1. *(20 + 7 points)* ***Monitors and semaphores***

a. *(10 points) An inefficient solution to the multiple producer/multiple consumer problem you solved in Program 2 might look like the pseudocode below, which uses a single lock, L, and a single condition variable, CV:*

Producer	*Consumer*
lock(L)	*lock L*
while (buffer full)	*while (buffer empty)*
wait(L, CV)	*wait(L, CV)*
Write data to buffer	*Read data from buffer*
signal(CV)	*signal(CV)*
unlock(L)	*unlock(L)*

Explain what makes this solution inefficient and how you can solve this issue.

Solution: The biggest issue is that using only a single condition variable means you have threads potentially signaling threads of the same type, not the opposite type (i.e., producers signaling consumers and vice-versa).

A smaller issue is that each thread signals every time it executes, as opposed to only signaling when the wait condition for the other thread has become false. In other words:

- A producer that writes to an empty buffer (changes the number of values in the buffer from 0 to 1) should signal a consumer; all other producers should not.
- A consumer that reads from a full buffer (changes the number of values in the buffer from its capacity to (capacity – 1)) should signal a producer; all other consumers should not.

1 (continued)

b. *(10 points) In Program 2, you implemented a monitor using locks and condition variables. Could you implement a monitor for the same problem (bounded buffer, multiple producers & consumers) using only semaphores for synchronization? If not, explain why not; if so, explain for what purposes you would need semaphores, how you would initialize those semaphores, and under what conditions you would call up() or down() on each of them.*

Solution: Since a semaphore can effectively replace a lock or condition variable, we can certainly create a monitor for this problem that only uses semaphores. You'd need three semaphores:

- One semaphore replaces the lock protecting access to the buffer. This semaphore should be initialized to 1; each thread should call down() before accessing the buffer and up() once that access is complete.
- One semaphore represents the number of empty slots in the buffer, so it should be initialized to the size of the array. Each producer should call down() on this semaphore prior to writing the buffer, so, if the buffer is full, the producer will block until a slot opens up. Each consumer should call up() on this semaphore after reading a value, thus effectively signaling a producer that a slot is free.
- One semaphore represents the number of full slots in the buffer, so it should be initialized to 0. Each consumer should call down() on this semaphore prior to reading the buffer, so, if the buffer is empty, the consumer will block until a value is written. Each producer should call up() on this semaphore after writing a value, thus effectively signaling a consumer that data is available to read.

c. *(7 points, **EECE.5730 only**) A barrier is a synchronization construct that forces a group of threads to wait for all threads to reach the barrier before allowing any of them to proceed. Use the space below to describe how a barrier can be implemented using semaphores for a given number of threads, N. (Hint: there are two general solutions—a relatively inefficient one using N semaphores, and a more efficient one using two semaphores and an integer.)*

Solution: As written above, there are two solutions:

- Use N semaphores, one associated with each thread, all initialized to 0. When a thread reaches the barrier, it calls up() on its own semaphore—signaling the other N-1 threads—and down() on the other N-1 semaphores to wait for them to reach the barrier as well.
- The efficient solution uses one semaphore as a mutex, which is initialized to 1, one semaphore for waiting, which is initialized to 0, and a shared integer to track the number of threads at the barrier, initialized to 0. Each thread does the following at the barrier:
    - Lock the "mutex" (call down()), increment the count, then unlock the mutex
    - If count == n, call up() on the waiting semaphore to signal the next thread
    - Call down() on the waiting semaphore, followed by up(), so that:
        - All threads wait to be woken—the Nth thread will wake up another thread, then wait itself.
        - Once each thread wakes up, it signals another thread.

2. *(30 points)* ***Deadlock***

a. *(20 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 total requested resources, initial state (resources the process is already using at this point), and a list of individual requests. Each individual 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* *3 connections*	*5 files* *5 connections*	*5 files* *3 connections*
Initial state (already using)	*1 file*	*2 connections*	*1 file*
Individual resource requests	*A1: 2 files, 1 conn.* *A2: 2 conn.* *A3: 1 file*	*B1: 4 files* *B2: 1 file, 1 conn.* *B3: 2 conn.*	*C1: 3 conn.* *C2: 2 files* *C3: 2 files*

Solution: This problem doesn't require you to interleave requests from different threads—in fact, if you try to do so, you'll get stuck—but it does impose a specific order between the three threads, because 2 files and 2 connections are in use when you start. Looking at the remaining resource requests for each thread:

- Thread A needs 3 more files and 3 more connections to finish.
- Thread B needs 5 more files and 3 more connections to finish.
- Thread C needs 4 more files and 3 more connections to finish.

The threads must therefore run in the order A, C, B. The next page shows a step-by-step solution:

Step-by-step solution to problem 2a:

Initial state: 2 files, 2 connections used

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

b. (10 points) You are running a program with three concurrent threads, each of which start execution by operating on two semaphores, S1 and S2, as shown. Assume both semaphores are initialized to 0:

Thread 1	*Thread 2*	*Thread 3*
`down(&S1);`	`down(&S1);`	`down(&S2);`
`up(&S2);`	`up(&S2);`	`down(&S2);`
…	…	`up(&S1);`
		`up(&S1);`
		…

Explain why these three threads are guaranteed to deadlock, and show how you could rewrite the threads to remove the deadlock condition.

Solution: Since every thread starts by calling down on a semaphore that's initialized to 0, the threads are guaranteed to deadlock. They'll all be stuck waiting for each other to increment one of the semaphores.

Resolving this deadlock means changing the order of at least one of the threads. Either:

- Thread 1 & 2 must both call `up(&S2)` before calling `down(&S1)`, or
- Thread 3 must complete both `up(&S1)` calls before calling `down(&S2)`

3. *(50 + 7 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	2	5	0
P2	7	4	0
P3	3	3	5
P4	8	2	7
P5	4	1	9

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 job first (SJF), and (iii) a preemptive priority scheme in which lower numbers indicate higher priority.

Your solution ***must contain some work that shows start/end times for each algorithm****—either a table or Gantt chart like the ones shown in class.*

You do not have to calculate the average turnaround time for each algorithm.

Solution: This problem is very similar to the example problem we did in Lecture 20. 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) SJF			(iii) Priority		
Proc	St	End	TT	St	End	TT	St	End	TT
P1	1	3	**3**	1	2	**2**	23	24	**24**
P2	2	18	**18**	3	9	**9**	1	22	**22**
P3	6	12	**7**	10	12	**7**	6	20	**15**
P4	9	24	**17**	17	24	**17**	8	19	**12**
P5	10	20	**11**	13	16	**7**	10	13	**4**

Detailed schedules for the two preemptive schemes (round robin and priority) 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	P2	**P1**	P2	P2	P3	P2	P3	P4	P5	P2	**P3**	P4	P5	P2	P4

Time	17	**18**	19	**20**	21	22	23	**24**
Process	P5	**P2**	P4	**P5**	P4	P4	P4	**P4**

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 for solving these problems by hand—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 priority scheduling, with the time in that process ran 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 7	8 9	10 13	14 19	20	21 22	23 24
Process (time)	P2 (1-5)	P3 (1-2)	P4 (1-2)	P5 (1-4)	P4 (3-8)	P3 (3)	P2 (6-7)	P1 (1-2)

3 (continued)

b. (10 points) When are scheduling decisions made in non-preemptive scheduling algorithms? When are scheduling decisions made in preemptive scheduling algorithms.

Solution: In non-preemptive schemes, scheduling decisions are made any time a process voluntarily leaves the CPU—the process finishes execution completely, or the process leaves to wait for an I/O device or other event.

In preemptive schemes, in addition to the points above (because these schemes do still make scheduling decisions when a process voluntarily gives up the CPU), scheduling decisions are made whenever a new process arrives, as that new process might preempt the currently running process.

c. (10 points) Which of the scheduling algorithms we discussed can cause starvation and why? How can you prevent starvation in these algorithms?

Solution: Any algorithm based on burst time (SJF, STCF) or priority can cause starvation, as longer jobs (in SJF/STCF) or higher priority jobs can indefinitely prevent other processes from gaining access to the CPU.

To prevent starvation, we use aging. In a priority scheme, aging is straightforward—the longer a process remains in the ready queue, the more we increase its priority, until it can't be ignored in favor of a higher priority job. In time-based schemes, we can adjust our burst time estimate to make it appear that a process will have a shorter burst time than it actually will.

3 (continued)

d. (10 points) Which scheduling algorithm would you choose for the set of processes described in each table? Briefly explain your answer in each case.

i.

Process	Burst	Arrival time
P1	10	0
P2	2	0
P3	15	1
P4	3	2
P5	8	3

Note: Any well-explained answer got at least some credit on this problem.

Solution: Given that some processes are significantly shorter than others, I'd use SJF or STCF to minimize wait time for those shorter jobs.

ii.

Process	Burst	Arrival time
P1	10	0
P2	10	0
P3	10	5
P4	9	7
P5	9	9

Solution: With all jobs having roughly equal burst times, FCFS scheduling would work just fine and have minimal scheduling overhead.

*e. (8 points, **EECE.5730 only**) In class, we described how some scheduling algorithms "degrade to first-come, first-served (FCFS)" under certain conditions. Explain what it means for an algorithm to degrade to FCFS, and give an example demonstrating this concept.*

Solution: "Degrading to FCFS" means that an algorithm will, under some circumstances, behave just like first-come, first-served scheduling, even though it's intended to use other metrics.

For example, SJF scheduling will degrade to FCFS if short jobs only after a longer job has been scheduled. If the scheduler's only choice is the longest possible job, that's what will be executed.