CSE 2431 **PRACTICE QUESTIONS (for 2nd half)** **SPRING 2025**

SOLUTION (To some questions)

Instructor: Luan Duong, Ph.D.

Acknowledgement: By doing these practice questions, you agree to the followings: Practice questions are **ONLY** for your practice. This set of practice questions **ONLY** serve **additional exercises** for you better prepare for your final. It is NOT a template as well. Also, the content of these questions **may not cover** the whole content of the final that you need to prepare for. Please follow the final review for the coverage. Please try out the questions on your own first before asking the TAs/. If you need additional questions, one thing you could do is to try changing the numbers from the previous questions or the lecture notes.

Q0: Important concepts: Again, please refer to the final review slides for what to expect for the finals. The **important** topics covered:

+ Memory structure: Virtual Memory: Dynamic Relocation, Segmentation, Paging and Swapping

+ Memory Hierarchy (Locality, Free memory managenent, Page replacement, Caching: Remember how to do direct mapped cache, E-way associative cache)

+ Concurrency: Locks, Condition Variables, Semaphores, Deadlocks (RAG), Classical Concurrency problems (Bounded Buffer, Reader-Writer, Dining Philosophers, Sleeping Barber)

+ I/O Devices: Disk, Disk Access time, Disk Scheduling, RAIDs, I/O devices (Interrupt, DMA)

+ Process/Thread Scheduling (FCFS/FIFO, STF, STCF, Preemptive, Non-preemptive, Round-robin, Multi-level Feedback Queue)

Q1: List all conditions for deadlock and write a one-line description of each.

Mutual Exclusion: Processes claim exclusive control over resources or locks.

Hold-and-Wait: Processes hold resources they have while waiting for other resources or locks.

No Preemption: The operating system cannot force a thread to release a resource or lock it holds.

Circular Wait: A circular chain of threads exists, where each thread is waiting for the next thread in the chain to release a resource or lock.

Q2: Complete the correct implementation of dining philosophers using semaphores. Your answers should either be down() or up().

```
Initialize semaphores chopstick[3] to 1 //Each chopstick is available

function philosopher(id):

      left = id                    // Philosopher's left chopstick

      right = (id+1)%3            // Philosopher's right chopstick

      while not done eating:

            print("Philosopher", id, "is thinking")

            //Pick up chopsticks

            if id is even then:

                  down(chopstick[left]);

                  down(chopstick[right]);

            else:

                  down(chopstick[right]);

                  down(chopstick[left]);

            Priont("Philosopher", id, "is eating")

            up(chopstick[left])

            up(chopstick[right])
```

Q4: What is swapping? Explain it in a few sentences and explain why we need it and how it works.

Swapping is the process of moving data out of volatile memory into persistent storage. It occurs when pressure on the volatile memory is high and has the effect of expanding the space of addressable memory.

Q5: Look at the following code snippets. Tell me if there is an error and if there is then give a brief description of how you may fix it:

Code 1:

```
pthread_mutex_t m = …

pthread_cond_t c = …

…

Thread0:                              Thread1:

      pthread_mutex_lock(&m)                pthread_mutex_lock(&m)

      pthread_cond_signal(&c)               pthread_cond_wait(&c, &m)

      …                                     …

      pthread_mutex_unlock(&m)              pthread_mutex_unlock(&m)
```

Code 2:

```
Thread0:                              Thread1:

      pthread_mutex_lock(&m)                pthread_mutex_lock(&d)

      pthread_mutex_lock(&d)                pthread_mutex_lock(&m)

      …                                     …

      pthread_mutex_unlock(&m)              pthread_mutex_unlock(&m)

      pthread_mutex_unlock(&d)              pthread_mutex_unlock(&d)

      …                                     …
```

Code 3:

```
pthread_mutex_t m = …

pthread_cond_t c = …

int trueOrFalse TF = …

…

Thread1:

      pthread_mutex_lock(&m)

            if (TF!=1){

                  pthread_cond_wait(&c, &m)

            }…
```

1. Usage of a condition variable should always be guarded by a condition to protect against spurious wakeups. Declare a third variable, int condition =0. Thread 1 should set condition =1 in the critical section. Thread 2 should wait with while (!condition) pthread_cond_wait(&c,&m);
2. This will deadlock if the first locks of each thread happen consecutively, as they will proceed to get stuck waiting on the next lock. They should always lock m and d in the same order.
3. This use of a condition variable does not guard against spurious wakeups. The "it" should be a "while".

Q6: Define each of the following components/terms (Check your lecture notes)

Sector:

Track:

Platter:

Head:

Seek Time:

Starvation:

Q7: Name and describe any deadlock prevention method.

Q8: Giving the following different cases, please state whether deadlock can happen, or not. If yes, draw the resource allocation graph (RAG) to describe and show the "cycle" for deadlock. Assume 1 lock = 1 type of resource

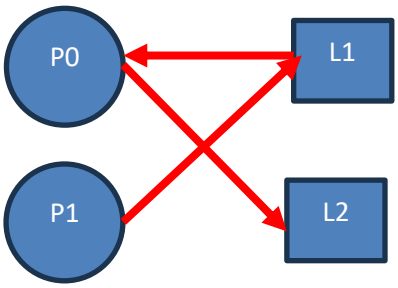
Case 1: [Code shown]

Process 0:	**Process 1:**
`lock1.acquire();`	`lock1.acquire();`
`lock2.acquire();`	`lock2.acquire();`
`lock1.release();`	`lock1.release();`
`lock2.release();`	`lock2.release();`

P0: try to grab lock 1 and lock 2 in order: 1 then 2.

P1: try to grab lock 1 and lock 2 in order: 1 then 2.

No Deadlock possible, because there is no circularity of requests. Both processes grab the locks in the same order. Here your RAG should be like this. We don't have arrow from P1 to L2 because P1 will be waiting for L1 to be released by P0 in this particular problem. No Cycle in RAG.



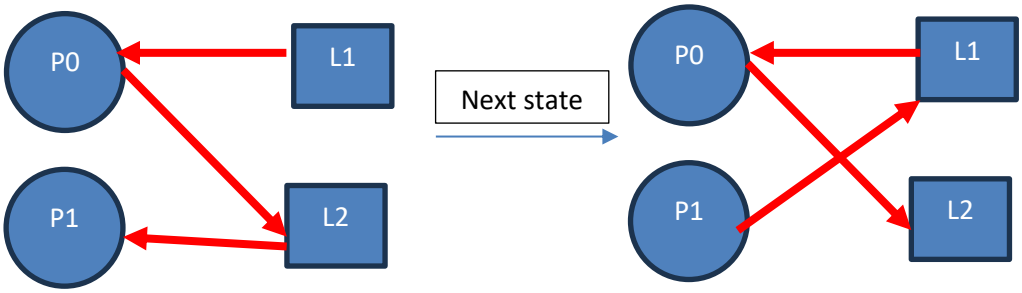
Case 2:

Process 0:	**Process 1:**
`lock1.acquire();`	`lock2.acquire();`
`lock2.acquire();`	`lock2.release();`
`lock1.release();`	`lock1.acquire();`
`lock2.release();`	`lock1.release();`

P0: try to grab lock 1 and lock 2 in order: 1 then 2.

P1: try to grab lock 2, then release it, then try to grab lock 1, then release it.

No deadlock possible, because process 1 does not hold and wait. Even if process 0 acquires lock1 and process 1 acquires lock2, eventually, process 1 will release lock2, at which point process 0 will be able to acquire lock 2. A sample RAG looks like this: Even L2 is given to P1, we do not need to have an arrow from P1 to L1 because we will have L2 release right after it is used by P1.



Q9 [Scheduling]: Giving the following processes with length and arrival time as follows:

Process	Length	Arrival time
P1	8	0
P2	4	0.4
P3	1	1.0

a. What is the turnaround time for each process with FCFS scheduling algorithm? What is the average turnaround time?
b. What is the response time for each process with SJF scheduling algorithm? What is the average response time?
c. What is the average response time for process with FCFS and SJF?
d. Do you notice any problem with this? Try to do b. again, but this time we choose to idle the CPU for the first unit and then schedule SJF again for those processes. Recalculate the average turnaround time and the average response time in this case.

Q10: Consider the following page reference string: 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacememnt algorithms, assuming one, two, three, four, five, six and seven frames (respectively)? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

1. LRU Replacement
2. FIFO Replacement
3. Optimal replacement.

Number of frames	LRU	FIFO	Optimal
1	20	20	20
2	18	18	15
3	15	16	11
4	10	14	8
5	8	10	7
6	7	10	7
7	7	7	7