University of Victoria
Midterm 2 PRACTICE QUESTIONS
Fall 2025

Student Name	
V-Number	
Section (CRN)	

Course Name & No.	CSC 360: Operating Systems
Instructor	Wenjun Yang
Duration	50 Minutes

- This exam has **4 questions** across 6 pages, including this cover page. Students must count the number of pages and report any discrepancy immediately.
- This exam is to be answered on the paper provided.
- A basic calculator may be used, although you should not need to use one.
- This is a closed-book exam, but a one-page cheat sheet is allowed.
- Ensure you do not look into cellphones during the exam. You must obtain permission from an invigilator to temporarily leave the examination room.
- We strongly recommend you read the entire exam through from beginning to end before beginning to write your answers.
- The total number of marks in this exam is **100**.
- **Please bring your ONECard for the ID check**.

1. CPU Scheduling Analysis (25 points)

Consider the following processes arriving at the times shown, with the given burst times and priorities (lower number = higher priority):

Process	**Arrival Time**	**Burst Time**	**Priority**
P1			
P2			
P3			
P4			

- Know how to calculate a variety of time (**waiting time, turnaround time**, etc) for algorithms: **FCFS**, **non-preemptive/preemptive SJF**, and **non-preemptive/ preemptive Priority Scheduling**.
- Understand the pros and cons of each scheduling algorithm.

Notes: In the real midterm, please give detailed steps and explanations on how you get the result. Thus, you can get part marks if you have a reasonable explanation even if the final result is wrong.

2. Sync Concepts (15 points)

Understand:

- mutex
- semaphores (binary vs counting)
- monitor
- condition variables

You may be asked about the differences among them and use cases for each of them.

Some students asked me about the difference between Mutex and Binary Semaphores. Here are hints and examples:

If you are incrementing or decrementing a variable, you should use a mutex. A mutex ensures that only one thread can access and modify the variable at a time, preventing data inconsistency.

For a semaphore, imagine going to a bank where there are multiple tellers providing service. Only one teller is available each time and each teller can help one customer. We can think of a binary semaphore named busy that indicates whether there is an available teller:

- busy = 0 means all tellers are busy,
- busy = 1 means a teller is free.

When customer 1 arrives, they acquire (wait on) the semaphore and get served by teller 1. Before teller 1 releases busy, teller 2 might become available, so teller 2 can signal (release) the semaphore to let customer 2 enter and get served.

Now, you should be able to summarize the differences in terms of ownership and purpose.

3. Sync Problems: Deadlock & Starvation (20 points)

Consider the following pseudocode:

```
Thread 1:                                   Thread 2:
lock(mutex_A)                               lock(mutex_B)
lock(mutex_B)                               lock(mutex_A)
// critical section                         // critical section
unlock(mutex_B)                             unlock(mutex_A)
unlock(mutex_A)                             unlock(mutex_B)
```

- Does this code have a potential **deadlock**? If yes, explain how.

Answer:

Yes, this code has potential deadlock.

Explanation: If the following interleaving occurs:
1. Thread 1 executes `lock(mutex_A)` - Thread 1 holds mutex_A
2. Thread 2 executes `lock(mutex_B)` - Thread 2 holds mutex_B
3. Thread 1 tries `lock(mutex_B)` - BLOCKS (Thread 2 holds it)
4. Thread 2 tries `lock(mutex_A)` - BLOCKS (Thread 1 holds it)

Now both threads are waiting for each other → **Deadlock**

- Provide ONE specific solution to prevent the deadlock.

Answer:

Necessary conditions present:
1. **Mutual Exclusion**: Mutexes are non-shareable
2. **Hold and Wait**: Each thread holds one mutex while waiting for another
3. **No Preemption**: Mutexes can't be forcibly taken away
4. **Circular Wait**: Thread 1 → mutex_B → Thread 2 → mutex_A → Thread 1

Solution (any ONE of the following):

1. Lock Ordering (Most common solution): Always acquire locks in the same order:

```
Thread 1:              Thread 2:
lock(mutex_A)          lock(mutex_A)  // Same order!
lock(mutex_B)          lock(mutex_B)
// critical section   // critical section
unlock(mutex_B)        unlock(mutex_B)
unlock(mutex_A)        unlock(mutex_A)
```

2. Try-Lock with Backoff:

```
Thread 2:
while (true) {
    lock(mutex_B)
    if (try_lock(mutex_A) == SUCCESS)
        break;
    unlock(mutex_B);  // Release and retry
    sleep(random_time);
}
```

3. Lock All at Once: Use a higher-level lock to acquire both atomically

4. The Real-World Sync Problem (40 points)

4.1. Problem Description

A barbershop located in Dragon Alley consists of a waiting room with **n** chairs plus an additional chair for the barber. If there are no customers to be served, the barber sits in his chair and goes to sleep. If a customer enters the barbershop and all **n** waiting-room chairs are occupied, then the (disappointed) customer leaves the shop. If the barber is busy cutting someone's hair, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep in his own chair, then the customer wakes up the barber.

4.2. Shared Data

The threads have available to them the following shared data with initial values:

```
int customers = 0;                    // Number of customers waiting
int n = 5;                            // Number of waiting chairs
semaphore mutex = Semaphore(1);       // Protects customers counter
semaphore customer_sem = Semaphore(0); // Barber waits on this
semaphore barber_sem = Semaphore(0);   // Customer waits on this
```

4.3. Complete the Implementation

Fill in the blanks in the following code. For each blank, write either `wait` or `signal` followed by the appropriate semaphore name (e.g., `wait(mutex)` or `signal(customer_sem)`).

Answers:

```
// Barber thread function
void barber() {
    while (true) {
        wait(customer_sem);          // Blank 1 & 2: Wait for customer to arrive

        wait(mutex);                 // Blank 3 & 4: Lock to modify counter

        customers--;                 // Blank 5: Update waiting customers

        signal(barber_sem);          // Blank 6 & 7: Signal readiness to customer

        signal(mutex);               // Blank 8 & 9: Release the lock

        cut_hair();                  // Perform the haircut
    }
}

// Customer thread function
void customer() {
    wait(mutex);                     // Blank 10 & 11: Lock to check conditions

    if (customers < n) {             // Check if chairs available

        customers++;                 // Blank 12: Update waiting customers

        signal(customer_sem);        // Blank 13 & 14: Wake up the barber
        signal(mutex);                // Blank 15 & 16: Release the lock

        wait(barber_sem);             // Blank 17 & 18: Wait for barber to be ready

        get_haircut();                // Get the haircut

    } else {
        signal(mutex);                // Blank 19 & 20: All chairs full, release lock
        // Customer leaves without haircut
    }
}
```

End of Exam