Review of Midterm Exam for COP 4610 Operating Systems - Fall 2024

Name: _____ **Student ID:** _____

Score: _____

♠ **This question is likely to be on the exam.**

HW1

1. (2 points) Having multiple processes running on a machine with multiple processors is an example of... (check all answers that apply)

   ☐ multitasking ☐ multithreading ☒ multiprogramming ☒ multiprocessing

2. (6 points) ♠ Match the terms on the right with the definitions on the left.

Definition	Term
1. A A general OS function	A. standard services
2. B A mechanism to protect one app from crashing another app	B. address space
3. D A unit of processing	C. time sharing system
4. F Collecting a batch of jobs before processing	D. job
5. C Multiple users can use terminals to interact with backend machines	E. dual-mode operations
6. E A mechanism to protect apps from crashing the OS	F. batch system

3. (1 point) A recipe is an analogy of ...

   ☒ a program ☐ neither

   ☐ both a program and a process ☐ a process

4. (2 points) How does a thread transition from the running state to the ready state? Check all answers that apply.

   ☒ a thread voluntarily yields ☒ a timer interrupt arrives

   ☐ an I/O request is completed ☐ a thread issues an IO request

5. (4 points) Match the terms on the right with the definitions on the left.

Definition	Term
1. C A sequential execution stream	A. program
2. B Containing all necessary states to run a program	B. address space
3. D An address space + at least one thread of execution	C. thread
4. A A collection of statements	D. process

   Please write the correct letter next to each number.

6. (1 point) Suppose 20% of a program can be executed in parallel. What is the maximum speedup for a computer with 5 cores?

Amdahl's Law: speedup $\leq \dfrac{1}{(1-P)+\dfrac{P}{N}}$, where P is the parallel portion and N is the number of cores.

☐ 1.04 ☐ 4.17 ☒ 1.19 ☐ 2.78

7. (1 point) ♠ Current trends in operating system kernel design favor a hybrid approach. This approach is based on which type of kernel, and what is the main reason driving this trend?

   ☐ Layered kernel, due to power efficiency.

   ☐ Modular kernel, due to development complexity.

   ☒ Microkernel, driven by the rise of distributed networks.

   ☐ Monolithic kernel, due to security concerns.

HW2

1. (7 points) ♠ During the boot sequence, what happens after each step? (Select the correct order for each step.)

   1. F jump to a fixed address in ROM
   2. D load an OS loader
   3. A load MBR (GPT) from the boot device
   4. B load the BIOS (UEFI)
   5. C load the kernel image
   6. F perform POST
   7. E set the kernel mode

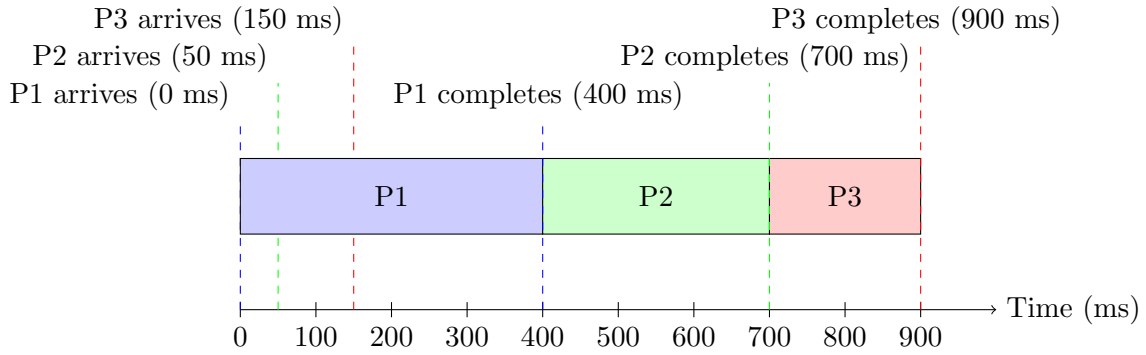
   A. load an OS loader
   B. perform POST
   C. set the kernel mode
   D. load the kernel image
   E. jump to the OS entry point
   F. load the BIOS (UEFI)
   F. load MBR (GPT) from the boot device

2. (1 point) Suppose we have three processes:

Process ID	Required CPU Time	Arrival Time
1	400 msec	0 msec
2	300 msec	50 msec
3	200 msec	150 msec

   For the FIFO scheduling, what is the turnaround time of process 3? Answer: 750

   ♠ FIFO Scheduling Diagram:



3. (1 point) For the FIFO scheduling, what is the wait time of process 1?

   Answer: 0

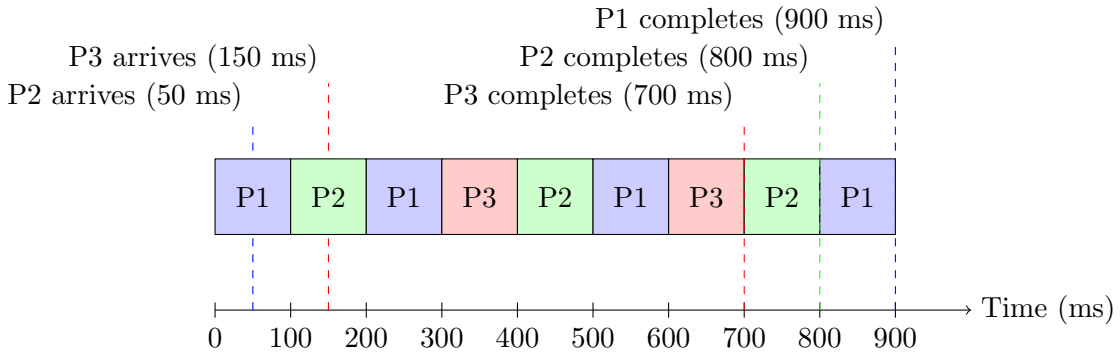
4. (1 point) For the FIFO scheduling, what is the wait time of process 3?

   Answer: 550

5. (1 point) Assume the time slice of 100 msec and a zero context-switching cost. For the round-robin scheduling, what is the wait time of process 3?

   Answer: 350

♠ **Round Robin Scheduling Diagram:**



Here is the breakdown of the sequence:

- At 49 ms, Process 1 is running.
- At 50 ms, Process 2 arrives, so the RR queue becomes [2].
- At 100 ms, Process 1 finishes its time slice, and the RR queue becomes [2, 1].
- At 100 ms, Process 2 starts running, and the RR queue is [1].
- At 150 ms, Process 3 arrives, so the RR queue becomes [1, 3].
- At 200 ms, Process 2 finishes its time slice, and the RR queue is [1, 3, 2].
- At 200 ms, Process 1 starts running, and the RR queue is [3, 2].
- At 300 ms, Process 1 finishes its time slice, and Process 3 starts running, with the RR queue now being [2, 1].
- At 400 ms, Process 3 finishes its time slice, and Process 2 starts running, with the RR queue being [1, 3].
- At 500 ms, Process 2 finishes its time slice, and Process 1 starts running, with the RR queue being [3, 2].
- At 600 ms, Process 1 finishes its time slice, and Process 3 starts running, with the RR queue being [2, 1].
- At 700 ms, Process 3 finishes and terminates. Process 2 starts running, with the RR queue being [1].

Thus, the wait time for Process 3 is calculated as:

$$\text{Wait time for Process 3} = (300 - 150) + (600 - 400) = 350 \text{ ms}$$

6. (1 point) Assume the time slice of 100 msec and a zero context-switching cost. For the round-robin scheduling, what is the response time of process 2?

   Answer: 50

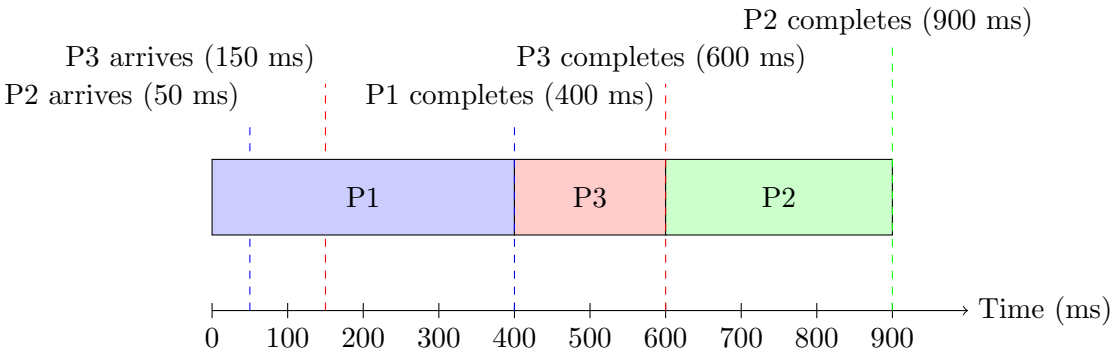
7. (1 point) Assume the time slice of 100 msec and a zero context-switching cost. For the round-robin scheduling, what is the turnaround time of process 1?

   Answer: 900

8. (1 point) For the SJF scheduling, what is the turnaround time of process 1?

   Answer: 400

   ♠ **SJF Scheduling Diagram:**



9. (1 point) ♠ For the SJF scheduling, what is the response time of process 2?

   Answer: 550

   Don't confused SJF (Shortest Job First) with SRJF (Shortest Remaining Job First).

   Someone believed that at 50 msec, Process 1 would have 350 msec Required CPU Time, while Process 2 would have 300 Required CPU Time, therefore Process 2 has the shortest time and will execute when it arrives, thus Process 2 will have 0 response time.

   However, at 50 msec, Process 2 has the shortest remaining time, not the shortest total time, which is why it would execute first when it arrives.

   For SJF scheduling, since it is non-preemptive, once a process starts running, it cannot be interrupted (unlike SRTF).

   Execution:

   - Process 1 runs from 0 to 400 msec.
   - Process 3 (shorter) runs from 400 to 600 msec.
   - Process 2 starts at 600 msec and finishes at 900 msec.

   Response Time for Process 2:

   $$\text{Turnaround time for Process 2} = 400\,\text{msec} + 200\,\text{msec} - 50\,\text{msec} = 550\,\text{msec}$$

10. (1 point) ♠ For the SJF scheduling, what is the turnaround time of process 2?

    Answer: 850

    For SJF scheduling, since it is non-preemptive, once a process starts running, it cannot be interrupted (unlike SRTF).

    Execution:

    - Process 1 runs from 0 to 400 msec.
    - Process 3 (shorter) runs from 400 to 600 msec.
    - Process 2 starts at 600 msec and finishes at 900 msec.

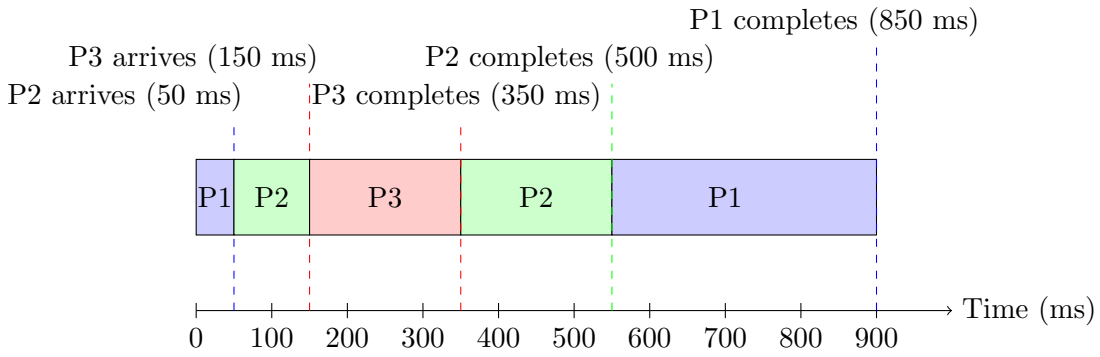
    Turnaround Time for Process 2:

    $$\text{Turnaround time for Process 2} = 900\,\text{msec} - 50\,\text{msec} = 850\,\text{msec}$$

11. (1 point) For the SRTF scheduling, what is the wait time of process 3?

    Answer: 200

    ♠ SRTF Scheduling Diagram:



12. (1 point) For the SRTF scheduling, what is the wait time of process 1?

    Answer: 500

13. (1 point) ♠ For the SRTF scheduling, what is the wait time of process 2?

    Answer: 200 or 0

    As you work through the SRTF (Shortest Remaining Time First) scheduling problem, you might encounter a situation where both Process 2 and Process 3 have equal remaining time. The question then arises: which one should you select?

    In real-world scenarios, typically Process 2 would continue running since it's already in execution. This approach minimizes the cost of a context switch. However, in our exercises, we assume that the context switch cost is zero. Therefore, you can choose either process, and both answers are considered valid.

    This means that for some questions, there may be two correct answers, depending on which process you choose to continue.

14. (4 points) Threads A and B are morning routines. Both threads share variables 'location' and 'action'.

    location = bedroom

    action = sleeping

```
// Thread A
1. enter the washroom
2. location = washroom
3. action = taking a break
4. enter the dining room
5. location = dining room
6. action = eating
```

```
// Thread B
7. enter the washroom
8. location = washroom
9. action = taking a break
10. enter the dining room
11. location = dining room
12. action = eating
```

    Assume that context switches can occur at any time. Is it possible that the shared variables can reach a state where 'location = washroom' and 'action = eating'? Please identify a sequence or sequences of operations that will lead to the embarrassing state.

    - ☒ 1, 7-12, 2
    - ☒ 1-5, 7-8, 6
    - ☒ 1-6, 7-8
    - ☐ 7-8, 1-6
    - ☐ It is not possible.
    - ☐ 1, 7, 2, 8, 3, 9, 4, 10, 5, 11, 6, 12

15. (2 points) Assuming both threads are running in the same address space, sharing variables 'x' and 'y', what are the possible final value pairs of 'x' and 'y'? Select all that apply.

```
// Thread A
x = 1;
x = x + 1;
```

```
// Thread B
y = 1;
y = x + 1;
```

- ☐ (3, 2)
- ☐ (1, 2)
- ☒ (2, 3)
- ☒ (2, 2)

HW3

1. (6 points) Match the terms on the right with the definitions on the left.

Definition	Term
1. __A__ An all or nothing operation	A. atomic operation
2. __B__ Results depend on the timing of executions	B. race condition
3. __D__ Using atomic operations to ensure cooperation among threads	C. mutual exclusion
4. __C__ Ensuring one thread can do something without the interference of other threads	D. synchronization
5. __E__ A piece of code that only one thread can execute at a time	E. critical section
6. __F__ Consuming CPU time while waiting	F. busy waiting

2. (1 point) A person uses a computer with a second-generation Intel i7 processor that has 2 physical cores and 4 logical threads to run a program that performs intensive calculations and requires relatively little memory access. Initially, they run the program with 1 thread, and it takes 60 seconds to complete. Then, they run the program with:

   2 threads: it finishes in 30 seconds.

   4 threads: it completes in 18 seconds.

   8 threads: it only improves slightly, finishing in 16 seconds.

   At first, increasing the thread count improved the execution time significantly, but the performance gains diminished when increasing the number of threads from 4 to 8. What is the most likely reason for this?

   ☐ There is a bug in the program that causes poor performance with more than 4 threads.

   ☐ The processor's cache is too small to support 8 threads efficiently.

   ☒ The CPU has only 2 physical cores and 4 logical threads, limiting the performance improvement with 8 threads.

   ☐ The operating system cannot handle more than 4 threads.

3. (2 points) ♠ Suppose variables x and y are both initially set to 0. Two threads run concurrently without any synchronization mechanisms:

```
// Thread A
x = 2;
x = x * y;
x = y + 1;
```

```
// Thread B
y = 1;
y = y + 2;
y = y * 2;
```

   After both threads have executed, which of the following results are possible? (Select all that apply.)

   ☒ x = 1, y = 6 ☒ x = 4, y = 6 ☐ x = 8, y = 6

   ☒ x = 7, y = 6 ☐ x = 3, y = 6 ☒ x = 2, y = 6

4. (2 points) ♠ Which of the following results occurs with the highest probability, given the same code from the previous question?

☐ x = 1, y = 6 ☒ x = 7, y = 6 ☐ x = 2, y = 6

☐ x = 8, y = 6 ☐ x = 4, y = 6 ☐ x = 3, y = 6

Explanation: There are 20 possible execution sequences when interleaving the instructions from both threads while maintaining the order within each thread. After analyzing all possible sequences:

x = 1, y = 6 occurs in 1 sequence.
x = 2, y = 6 occurs in 3 sequences.
x = 4, y = 6 occurs in 6 sequences.
x = 7, y = 6 occurs in 10 sequences.
Therefore, the result x = 7, y = 6 occurs with the highest probability.

5. (1 point) In multithreaded programming, race conditions are a common issue. For example, using 'x++' to increment a global variable 'x' may lead to incorrect results because the operation is not atomic (it involves load, compute, and store steps). Meanwhile, 'printf' is a more complex operation than 'x++'. In a multithreaded program where two threads print the characters 'a' and 'b' respectively, which of the following statements is correct?

   ☒ The program's output may contain unexpected or corrupted characters because 'printf' is a complex operation, and calling it from multiple threads may lead to unpredictable behavior.

   ☐ The program's output will contain only 'a' and 'b' without any errors or unexpected characters, because the operating system treats 'printf' as thread-safe and ensures there is no data corruption.

6. (1 point) ♠ Consider a global variable 'x' initialized to 0. Two threads are each tasked with incrementing 'x' one million times. Four different implementations are proposed to perform the increment operation. After both threads have completed their execution, which implementation will consistently result in 'x' being exactly 2,000,000?

   ☐ Using the assembly instruction 'asm volatile("addq $1, %0" : "+m"(x));' to increment 'x' (without any locking mechanism).

   ☒ Using the assembly instruction 'asm volatile("lock addq $1, %0" : "+m"(x));' to increment 'x' (with the 'lock' prefix to ensure atomicity).

   ☐ Using 'x++' to increment 'x' in each thread.

   ☐ Implementing a simple lock by using a shared flag variable to protect 'x++', where each thread checks the flag before proceeding with the increment (without proper atomic operations or memory barriers).

7. (1 point) ♠ Consider a global variable 'x' initialized to 0. Two threads are each tasked with incrementing 'x' one million times. Four different implementations are proposed to perform the increment operation. After both threads have completed their execution, which implementation will take the longest time to complete?

   ☐ Using the assembly instruction 'asm volatile("addq $1, %0" : "+m"(x));' to increment 'x' (without any locking mechanism).

   ☐ Implementing a simple lock by using a shared flag variable to protect 'x++', where each thread checks the flag before proceeding with the increment (without proper atomic operations or memory barriers).

   ☒ Using the assembly instruction 'asm volatile("lock addq $1, %0" : "+m"(x));' to increment 'x' (with the 'lock' prefix to ensure atomicity).

   ☐ Using 'x++' to increment 'x' in each thread.

8. (1 point) In the file 'lecture 6's ex_6_spin_scalability.c', a multi-threaded approach is used to increment a shared variable 'x' by splitting the workload across multiple threads. The synchronization mechanism used in the code depends on the compilation flag: either a spinlock ('spin_lock()') or a mutex ('mutex_lock()') is used. You run this program with an increasing number of threads (e.g., 2, 4, 8, 16, 32) and notice that the performance does not scale well as the number of threads increases. What would be the most effective way to improve the scalability and efficiency of the program?

☐ Reduce the number of threads to only one, as multi-threading introduces synchronization overhead that can slow down execution.

☐ Increase the number of iterations (N) to allow each thread to do more work, which reduces the synchronization overhead.

☒ Replace the locking mechanism with an atomic operation, such as 'atomic_fetch_add(&x, 1)', eliminating the need for locks altogether.

☐ Use a mutex instead of a spinlock, as a mutex allows threads to sleep when they can't acquire the lock, avoiding busy-waiting.

HW4

1. (1 point) ♠ Which of the following statements about spinlocks is correct?

   ☐ Spinlocks are more efficient than mutexes in all scenarios.

   ☒ A spinlock uses busy waiting, meaning a thread repeatedly checks the lock without releasing the CPU.

   ☐ Spinlocks can be used to avoid busy waiting.

   ☐ A spinlock allows multiple threads to access the critical section simultaneously.

2. (1 point) ♠ What is the main benefit of using the 'xchg' instruction in the 'acquire_lock()' implementation shown below?

```
int lock = 0;
int xchg(int *addr, int newval) {
    int result;
    asm volatile (
    "lock xchg %0, %1"
    : "+m" (*addr), "=a" (result)
    : "1" (newval)
    : "cc"
    );
    return result;
}

void acquire_lock(int *lock) {
    while (xchg(lock, 1)) {}
}

void release_lock(int *lock) {
    xchg(lock, 0);
}

void *foo(void *arg) {
    acquire_lock(&lock);
    // Critical section: Do work here ...
    release_lock(&lock);
    return NULL;
}
```

   ☐ It allows multiple threads to enter the critical section at once.

   ☐ It ensures that the lock is released automatically after a fixed time.

   ☒ It performs the lock acquisition as an atomic operation, preventing race conditions.

   ☐ It allows the lock to be acquired without the use of hardware support.

3. (1 point) ♠ What does the ‘xchg‘ instruction do in the context of acquiring a lock?

   ☒ It swaps the values of the lock and the result atomically, ensuring no other thread can modify the lock during this operation.

   ☐ It checks if the lock is available and waits until it is free without swapping any values.

   ☐ It releases the lock by resetting the value to 0.

   ☐ It increments the value of the lock each time it is called.

4. (1 point) Why is the "acquire_lock()" implementation shown below potentially problematic in a multi-core system?

```
int lock = 0;
void acquire_lock(int *lock) {
    while (*lock != 0) {}
    *lock = 1;
}
void release_lock(int *lock) {
    *lock = 0;
}

void *foo(void *arg) {
    acquire_lock(&lock);
    // Critical section: Do work here ...
    release_lock(&lock);
    return NULL;
}
```

   ☒ It allows race conditions because multiple threads can modify the lock without atomic operations.

   ☐ It uses too many atomic operations, which decreases performance.

   ☐ It prevents threads from releasing the lock once they acquire it.

   ☐ It always acquires the lock without checking if the lock is available.

5. (1 point) ♠ What happens to threads waiting for a mutex when the lock is already held by another thread?

   ☐ They continue busy-waiting until the lock is released.

   ☒ They enter a blocked state and the OS puts them to sleep until the lock becomes available.

   ☐ They are terminated by the OS to prevent deadlocks.

   ☐ They use atomic instructions to steal the lock from the current holder.

6. (1 point) What is a futex and how does it improve performance compared to a spinlock?

   ☐ A futex allows multiple threads to hold the lock at the same time.

   ☐ A futex is a special type of mutex that always uses the kernel to manage thread sleep and wake-up.

   ☒ A futex combines the advantages of spinlocks and mutexes by starting with spinning and escalating to a kernel-based mutex.

   ☐ A futex is a user-space only lock that prevents any kernel intervention.

7. (1 point) ♠ In a mutex-based system, how does the OS manage threads waiting for a lock?

   ☐ The OS puts all waiting threads in a busy-wait loop until the lock is released.

   ☐ The OS allows only the highest-priority thread to acquire the lock.

   ☒ The OS puts the threads into a blocked (sleep) state and wakes them up when the lock is released.

☐ The OS continually switches between all waiting threads, giving each a chance to acquire the lock.

8. (1 point) In a scenario where three threads (X, Y, and Z) are waiting for a mutex, what typically happens when the lock is released by Thread X?

   ☐ Both Thread Y and Thread Z are woken up, and they compete for the lock.

   ☐ Thread Z is woken up first if it has a higher priority than Thread Y.

   ☐ The OS randomly selects one of the waiting threads (Y or Z) to wake up.

   ☒ The OS wakes up Thread Y, following a first-come, first-served (FIFO) or priority-based policy.

9. (1 point) Why is the futex mechanism often more efficient than using pure spinlocks or pure mutexes?

   ☒ It uses a combination of spinning and sleeping, reducing busy-waiting and unnecessary context switches.

   ☐ It allows threads to share the lock when there is high contention.

   ☐ It automatically prioritizes the most important threads.

   ☐ It prevents context switches entirely.

10. (1 point) ♠ What is the primary goal of the Producer-Consumer problem?

    ☐ To ensure that consumers consume data faster than producers can produce.

    ☐ To avoid the need for condition variables or semaphores.

    ☒ To ensure synchronization between producers and consumers, so that the buffer doesn't overflow or underflow.

    ☐ To ensure that producers produce data as fast as possible, regardless of the buffer size.

11. (1 point) ♠ What is the major disadvantage of using a single condition variable in the Producer-Consumer problem?

    ☐ It allows both producers and consumers to operate simultaneously without synchronization.

    ☐ It improves performance by reducing the number of context switches.

    ☒ It can lead to deadlock if the same type of thread (producer or consumer) is repeatedly woken up.

    ☐ It makes the program easier to debug.

12. (1 point) Why does busy-waiting in the implementation of the Producer-Consumer problem shown below waste CPU resources?

```
void *Tproduce(void *arg) {
    while (1) {
        retry:
        pthread_mutex_lock(&lk);
        if (count == n) {
            pthread_mutex_unlock(&lk);
            goto retry;
        }
        count++;
        printf("O");
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}

void *Tconsume(void *arg) {
    while (1) {
```

```
        retry:
        pthread_mutex_lock(&lk);
        if (count == 0) {
            pthread_mutex_unlock(&lk);
            goto retry;
        }
        count--;
        printf("X");
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

☐ Because it uses semaphores incorrectly.

☐ Because it only works on single-core systems.

☐ Because it doesn't allow threads to enter the critical section simultaneously.

☒ Because it repeatedly checks for a condition without yielding the CPU, even when conditions are not met.

13. (1 point) ♠ How do condition variables solve the busy-waiting problem in the Producer-Consumer problem?

☐ They guarantee that producers and consumers work in parallel.

☐ They eliminate the need for synchronization entirely.

☒ They allow threads to wait until a condition is met without consuming CPU resources.

☐ They prevent multiple threads from entering the critical section.

14. (1 point) In the following scenario with a buffer size of n=1:

The buffer is initially empty, and both consumers (C1, C2) are sleeping. Producer P2 is sleeping because the buffer was previously full. Producer P1 produces an item, filling the buffer, and signals a 'buffer change' event. The signal wakes up P2, but P2 finds the buffer still full, so P2 goes back to sleep without signaling. The OS schedules P1 again, but P1 also finds the buffer full and goes back to sleep without signaling.

After P1 is rescheduled by the OS and goes back to sleep without signaling, which thread will the OS likely schedule next, and can this thread signal the waiting consumers (C1, C2)? Why or why not?

☐ P2 will be scheduled and will signal the waiting consumers (C1, C2) even if it cannot proceed itself.

☐ One of the consumers (C1 or C2) will be scheduled, and it will signal the producers after consuming the item.

☒ The OS will not signal any thread, leading to deadlock because both producers and consumers are stuck waiting for signals.

☐ P2 will be scheduled again, but it cannot signal consumers because the buffer is still full.

15. (1 point) Why does using two condition variables (not_full and not_empty) prevent deadlock in the Producer-Consumer problem?

☒ Because producers signal only consumers and vice versa, ensuring that progress is always made by at least one thread type.

☐ Because the buffer size is always kept constant.

☐ Because it prevents producers from producing more items than consumers can consume.

☐ Because the OS automatically ensures that only one type of thread runs at a time.

16. (1 point) In the context of deadlock, what role does the operating system play in thread synchronization?

☐ The OS manages thread scheduling and CPU time allocation, but does not manage thread synchronization or signal passing.

☐ The OS automatically handles all synchronization between threads, eliminating the need for condition variables.

☒ The OS automatically signals threads waiting for a condition to be met.

☐ The OS ensures that no thread can enter the critical section if another thread is already there.

17. (1 point) ♠ Which of the following best describes the role of 'pthread_cond_wait' in a condition variable-based synchronization mechanism?

☐ It immediately acquires the lock, regardless of whether the condition is met.

☐ It wakes up all waiting threads once the condition is met.

☒ It releases the mutex and waits for the condition to be signaled before reacquiring the mutex.

☐ It ensures that threads remain active while waiting for the condition.

18. (1 point) What is the main difference between a semaphore and a condition variable?

☐ Semaphores are always used with mutexes, whereas condition variables do not require mutexes.

☒ Semaphores allow multiple threads to access the critical section simultaneously, whereas condition variables typically allow only one thread at a time.

☐ Condition variables are used in user space, whereas semaphores are only used in kernel space.

☐ Condition variables automatically track resource availability, whereas semaphores do not.

19. (1 point) ♠ What does the 'P' operation in semaphores do?

☒ Decreases the semaphore's value by 1 and may block the thread if the value is negative.

☐ It checks if the buffer is full and puts the thread in a busy waiting loop.

☐ Puts the thread to sleep until the semaphore's value becomes positive.

☐ Increases the semaphore's value by 1 and signals a thread to continue.

20. (1 point) ♠ In the Producer-Consumer problem, what is the main advantage of using semaphores without mutexes when dealing with simple operations?

☐ It simplifies the code, making it easier to implement complex buffer operations.

☐ It allows threads to produce and consume items even when the buffer is full or empty.

☐ It prevents deadlocks from occurring.

☒ It reduces overhead by eliminating the need for locking and unlocking, allowing more threads to operate concurrently.

21. (1 point) ♠ In which scenario is using condition variables preferable over semaphores?

☐ When multiple threads need to access the critical section simultaneously.

☒ When the synchronization logic becomes more complex, and you need finer control over waiting and signaling.

☐ When you want to avoid using any locking mechanisms in your program.

☐ When you have a simple, static buffer that does not require complex operations.

22. (1 point) What is the primary focus of High-Performance Computing (HPC)?

☐ Handling system calls efficiently.

☒ Task decomposition and parallel computation.

☐ Managing thread context switching.

☐ Simplifying user interaction with web pages.

23. (1 point) Which technology is commonly used in High-Performance Computing for parallel processing?

☐ JavaScript.

☐ Goroutines.

☒ OpenMP.

☐ Promise.

24. (1 point) What is the main challenge in breaking down computational tasks in HPC?

☐ Managing shared memory access.

☐ Handling network requests asynchronously.

☐ Developing user-friendly interfaces.

☒ Ensuring that computation graphs are easy to parallelize.

25. (1 point) What is the primary focus in concurrent programming within data centers?

☐ Managing graphical user interfaces.

☒ Handling system calls efficiently and serving massive requests.

☐ Task decomposition for parallel processing.

☐ Using promises for asynchronous operations.

26. (1 point) What is a key reason why coroutines do not require context switching?

☒ They are scheduled in user space without kernel involvement.

☐ They rely on OS-level thread management.

☐ They automatically handle all system calls.

☐ They are only used in HPC environments.

27. (1 point) What is the main benefit of using Goroutines in Go programming for data centers?

☐ Goroutines allow sequential task execution.

☒ Goroutines combine the lightweight nature of coroutines and the power of threads.

☐ Goroutines do not require memory management.

☐ Goroutines are faster than JavaScript promises.

28. (1 point) Which of the following best describes the benefit of using asynchronous callbacks in JavaScript?

☐ Each asynchronous operation blocks the execution until it is completed, ensuring tasks are performed in sequence.

☐ Asynchronous operations are limited to network requests and cannot be used for other types of tasks.

☒ Asynchronous operations immediately return when started, and a callback function is called when the operation completes, allowing the program to continue executing other tasks without waiting for the asynchronous operation to finish.

☐ Asynchronous callbacks are slower than traditional synchronous programming due to the overhead of managing multiple threads.

29. (1 point) Which technology primarily powers asynchronous event-driven programming in Web 2.0?

☒ Promise.

☐ Goroutines.

☐ OpenMP.

☐ MPI.

30. (1 point) What is the main advantage of using promises in JavaScript?

    ☒ They prevent callback hell by improving code readability.
    ☐ They allow direct parallelism.
    ☐ They use hardware-level atomic instructions.
    ☐ They are better suited for numerical computation.

HW5

1. (1 point) Which of the following is NOT a condition necessary for a deadlock to occur?

    ☒ Preemptable resources
    ☐ Circular wait
    ☐ No preemption
    ☐ Wait while holding

2. (1 point) In the Dining Philosophers problem, what would lead to a deadlock if all philosophers follow the same sequence of actions?

    ☒ Philosophers grab the right chopstick before the left one at the same time.
    ☐ Philosophers grab the left chopstick before the right one.
    ☐ Philosophers only eat after thinking.
    ☐ Philosophers start eating before they grab chopsticks.

3. (1 point) Which of the following deadlock prevention techniques involves making sure no thread waits while holding a resource?

    ☒ No waiting
    ☐ Circular wait prevention
    ☐ Infinite resources
    ☐ Banker's Algorithm

4. (1 point) ♠ In the following scenario, how many threads are required to cause a deadlock?

```
void os_run() {
    spin_lock(&list_lock);
    spin_lock(&xxx);
    spin_unlock(&xxx);
}
void on_interrupt() {
    spin_lock(&list_lock);
    spin_unlock(&list_lock);
}
```

    ☒ 1 ☐ 2 ☐ 5 ☐ 3

5. (1 point) ♠ What is a practical approach to defensive programming?

    ☐ Ignoring potential edge cases
    ☒ Using assertions to validate program conditions
    ☐ Focusing only on speed optimizations
    ☐ Implementing all features before testing