Do not turn this page until you have received the signal to start.

(Please fill out the identification section above, write your name on the back of the test, and read the instructions below.)

Good Luck!

This midterm consists of 5 questions on 10 pages (including this one). When you receive the signal to start, please make sure that your copy is complete. Pseudo-code is acceptable where code is required. Answer the questions concisely and legibly. Answers that include both correct and incorrect or irrelevant statements will not receive full marks. If you use any space for rough work, indicate clearly what you want marked.

TOTAL: _____/32
Question 1. Multiple Choice - Mark the best choice for each part [10 marks]

Part (a) [1 mark] Which of the following operations must be privileged?

- [ ] Modifying the stack pointer register (SP)
- [ ] Modifying the program counter register (PC)
- [ ] Modifying the page table base register (PTBR)
- [ ] Enabling interrupts
- [ ] All of the above

Part (b) [1 mark] System calls...

- [ ] are exactly like function calls, except for switching the CPU to privileged mode.
- [ ] allow user-level processes to request OS services.
- [ ] are the only mechanism to enter the OS when a user-level process is executing.
- [ ] use the process’s user-level stack for local variables while executing in the OS.
- [ ] All of the above.

Part (c) [1 mark] A system has \( N \) concurrently executing processes. In normal operation (i.e., not a temporary intermediate state during a context switch), what is the maximum number of processes that can be in the Ready state?

- [ ] \( N \)
- [ ] \( N - 1 \)
- [ ] \( N - 2 \)
- [ ] 1
Part (d) [1 mark] A multi-threaded program may be implemented with either a pure user-level threads package or a kernel-level threads package. In which of the following cases would the user-level threads package be better?

- The program frequently creates and switches between threads.
- The program frequently issues `read()` system calls.
- The program is solving a parallel problem and runs on a multiprocessor.
- The program would like to prevent threads from modifying values on the stacks of other threads.
- All of the above.

Part (e) [1 mark] A race condition happens when

- Two threads share a critical section and one of the threads is never able to enter the critical section while the other thread enters it repeatedly.
- Two threads share a critical section and one of the threads is not able to enter the critical section even though the other thread is not executing the entry, exit, or critical section code itself.
- Two threads access a shared resource and it is impossible to tell which one executed first.
- Two threads access a shared resource without synchronization and the outcome depends on the order of execution.
- None of the above.

Part (f) [1 mark] A thread that is blocked on a semaphore is awakened when another thread

- Tries to decrement the semaphore’s value below 0.
- Tries to increment the semaphore.
- Causes the semaphore’s value to reach a specific number.
- Tries to block on the same semaphore.
Part (g)  [1 mark] Which of the following is NOT a problem for spinlocks?

- [ ] CPU cycles are wasted due to busy waiting.
- [ ] Starvation is possible.
- [ ] Non-portable code due to use of hardware atomic instructions.
- [ ] Cannot be used on a multiprocessor.
- [ ] Cannot be implemented at user-level.

Part (h)  [1 mark] You are running a process that creates 1000 directories (directory creation requires a blocking I/O operation) on a system with a large number of other processes. Which CPU scheduling algorithm would you prefer?

- [ ] FCFS
- [ ] Round Robin
- [ ] Multi-level feedback queues
- [ ] It doesn’t matter to you, you just care about disk speed not CPU.

Part (i)  [1 mark] A performance problem with the Round Robin scheduler compared to FCFS is:

- [ ] CPU cycles are wasted when a process blocks for I/O before its quantum expires.
- [ ] CPU cycles are wasted due to more frequent context switches.
- [ ] Interactive response time is poor.
- [ ] Starvation is possible.

Part (j)  [1 mark] A particular architecture supports paged memory translation using 16-bit virtual addresses with a 1024 byte page size. What is the maximum number of virtual pages that a process on this system can have?

- [ ] \(2^4 = 16\)
- [ ] \(2^6 = 64\)
- [ ] \(2^8 = 256\)
- [ ] \(2^{10} = 1024\)
Question 2.  True / False [6 marks]

For each statement, circle the correct option.

TRUE  FALSE  Context switching is handled in hardware.

TRUE  FALSE  Multi-threading is useful even on a uniprocessor.

TRUE  FALSE  The C library function malloc requires a system call to allocate memory on every invocation.

TRUE  FALSE  Monitors with MESA semantics can support a wake-all-waiting (broadcast) operation on a condition variable, but with Hoare semantics they cannot.

TRUE  FALSE  Operating system code can disable interrupts to make short critical sections atomic on multiprocessors.

TRUE  FALSE  Starvation is impossible if a pre-emptive scheduling algorithm is used.
Question 3. Very Short Answer [7 marks]

Each of the parts below should be answered with a few words, or one sentence at most.

Part (a) [1 mark] Why does the OS typically maintain multiple queues to keep track of threads in the blocked state?

Part (b) [1 mark] The operating system maintains a data structure to represent each thread. List two items (give a general description, not specific variable names) that would appear in this structure.

Part (c) [1 mark] How does the fork() system call return twice?

Part (d) [1 mark] Which CPU scheduling algorithm is optimal with respect to average wait time?

Part (e) [3 marks] Circle the best choice in each capitalized set in the following statements.

C library heap management routines (e.g. malloc and free) implement a \{FIXED | DYNAMIC\} partitioning scheme.
The heap can suffer from \{INTERNAL | EXTERNAL | BOTH TYPES OF\} fragmentation.
Compaction \{CAN | CANNOT\} be used to reduce fragmentation in the heap.”
Question 4. Concurrency [3 MARKS]

Suppose a program creates several threads that concurrently execute a loop in which they first increment and then decrement a shared variable `counter` without any synchronization, as follows:

```c
for (i = 0; i < nites; i++) {
    counter++;
    counter--;
}
```

We observe the following: When `nites=1000`, on a uniprocessor the final value of `counter` is almost always the same as its initial value regardless of the number of threads used, but on a multiprocessor different final values for `counter` are frequently observed even with only 2 threads. **Explain this result.**

---

This space intentionally left blank.
Question 5. Synchronization [6 marks]

You are asked to conduct a code review of the implementation of a user-level semaphore synchronization primitive. The design goals require that:

1. The `sem_wait()` operation must **not** require a system call if the semaphore is available and no other thread is trying to obtain it concurrently. (This is called the *uncontested* case.)

2. The `sem_post()` operation must **never** require a system call.

3. The `sem_wait()` operation should put the thread to sleep for some amount of time when the semaphore is unavailable to reduce the amount of busy waiting.

A user-level `spinlock_acquire()` / `spinlock_release` primitive built on a hardware test-and-set atomic instruction is available.

Here is the code you are asked to review:

```c
typedef struct sem_s {
    spinlock_t lock;
    int count;
} sem_t;

void sem_init(sem_t *sem, int val) {
    spinlock_init(&sem->lock);
    sem->count = val;
}

void sem_post(sem_t *sem) {
    spinlock_acquire(sem->lock);
    sem->count ++;
    spinlock_release(sem->lock);
}

void sem_wait(sem_t *sem) {
    int acquired = 0;
    while (!acquired) {
        spinlock_acquire(sem->lock);
        if (sem->count == 0) {
            /* ms_sleep is a system call that puts a process to sleep
             * for some number of milliseconds
             */
            ms_sleep(1);
        } else {
            sem->count --;
            spinlock_release(sem->lock);
            acquired = 1;
        }
    }
}
```

Answer the questions on the following page with reference to this code.
Question 5. Synchronization (CONTINUED)

Part (a) [2 Marks] Ignoring any coding errors, does the code appear to meet the design goals? Explain briefly.

Part (b) [2 Marks] Ignoring any coding errors, would this implementation guarantee that sem_wait() requests are granted in the order in which they arrive? Explain briefly.

Part (c) [2 Marks] You wisely reject the code because you spot a fatal mistake in the sem_wait() implementation that will lead to deadlock. Briefly explain the problem and how it should be fixed.