Exam 1

601.418/618 Operating Systems

March 3, 2025

Complete all questions.

Time: 75 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: ______________________________

Print name: ______________________________

Date: ______________________________

Synchronization reference:

```
// Mutex lock operations
void lock_init( lock_t *lock );
void lock_acquire( lock_t *lock );
void lock_release( lock_t *lock );

// Semaphore operations
void sem_init( sem_t *sem, int count );
void sem_wait( sem_t *sem ); // P() operation
void sem_post( sem_t *sem ); // V() operation

// Condition variable operations
void cond_init( cond_t *cond );
void cond_wait( cond_t *cond, lock_t *mutex );
void cond_signal( cond_t *cond );
void cond_broadcast( cond_t *cond );
```

Question 1. [10 points] For each of the following program operations, answer "yes" if the operation should be a privileged operation (executable only by code running in kernel mode), or "no" if the operation is safe to allow in code running in user mode. For the cases where you answered "yes", briefly explain why the operation shouldn't be allowed in user mode.

Operation	Privileged? (yes/no)	Explanation
Disable interrupts		
Read from a hardware I/O register		
Move a value to the stack pointer register		
Read the CPU timestamp register		
Execute `ret` (return from function)		

Question 2. [10 points]

Processes:

Process	Arrival	CPU time
0	1	6
1	5	4
2	6	1
3	9	6

Definitions:

The *wait time* of a process is
$T_{start} - T_{arrival}$

The *turnaround time* of a process is
$T_{finish} - T_{arrival}$

(a) Determine the start time, finish time, wait time, and turnaround time of each process assuming *First Come, First Served* (FCFS) scheduling. Assume that once a process begins executing it continues until it has finished. Drawing a diagram will be helpful (you can use to opposite page for scratch work if necessary.)

Process	Start time	Finish time	Wait time	Turnaround time
0				
1				
2				
3				

(b) What is the average wait time and average turnaround time for the above processes using FCFS scheduling? You may express your answer as a fraction.

Average wait time:

Average turnaround time:

[Use this page for scratch work if necessary for Question 2.]

Question 3. [10 points]

Processes:

Process	Arrival	CPU time
0	1	6
1	5	4
2	6	1
3	9	6

Definitions:

The *wait time* of a process is
$T_{start} - T_{arrival}$

The *turnaround time* of a process is
$T_{finish} - T_{arrival}$

(a) Determine the start time, finish time, wait time, and turnaround time of each process assuming *Shortest Job First* (SJF) scheduling. Assume that once a process begins executing it continues until it has finished. Drawing a diagram will be helpful (you can use to opposite page for scratch work if necessary.)

Process	Start time	Finish time	Wait time	Turnaround time
0				
1				
2				
3				

(b) What is the average wait time and average turnaround time for the above processes using SJF scheduling? You may express your answer as a fraction.

Average wait time:

Average turnaround time:

[Use this page for scratch work for Question 3.]

Question 4. [10 points]

Processes:

Process	Arrival	CPU time
0	2	6
1	6	4
2	7	2

Definitions:

The *wait time* of a process is
$T_{start} - T_{arrival}$

The *turnaround time* of a process is
$T_{finish} - T_{arrival}$

(a) Determine the start time, finish time, wait time, and turnaround time of each process assuming *Round Robin* (RR) scheduling. Assume that the scheduling quantum is 1 (i.e., time slices are one unit of time in duration.) Also, assume that if a process's arrival time is k, it starts just *before* time k. (For example, if a process arrives at time 3, it is ready to execute just *before* the time slice starting at time 3.) Drawing a diagram will be helpful (you can use to opposite page for scratch work.)

Process	Start time	Finish time	Wait time	Turnaround time
0				
1				
2				

(b) What is the average wait time and average turnaround time for the above processes using RR scheduling (with a quantum of 1)?

Average wait time:

Average turnaround time:

[Use this page for scratch work for Question 4.]

Question 5. [10 points] On x86 CPUs, when an exception occurs (trap, fault, or external interrupt), interrupts are automatically disabled before beginning the execution of the kernel interrupt routine for the specific exception type.

Why is it necessary for interrupts to be disabled when handling an exception? State an undesirable behavior that could happen if interrupts were enabled when starting the execution of the interrupt handler.

Question 6. [10 points] The *quantum* used by a scheduler is the amount of time a task can execute on a CPU before its time slice ends.

(a) State one advantage of a making the quantum relatively small.

(b) State one advantage of making the quantum larger.

Question 7. [10 points] On the right is a mutex lock design from Lecture 6. Recall that the `test_and_set` operation atomically stores a 1 in a memory location and returns the original value stored in that memory location. Assume that this lock implementation will be used to protect data shared by multiple threads.

A limitation of this lock design is that the `aquire` function consumes CPU time if the lock is currently held by another thread.

```
struct Lock {
  int held = 0;
};

void acquire(struct Lock *lock) {
  while (test_and_set(&lock->held));
}

void release(struct Lock *lock) {
  lock->held = 0;
}
```

(a) Briefly explain why this lock design would be undesirable on a single core system, but might be more useful on a multicore system.

(b) Briefly explain why the implementation of `acquire` on the right is better than the original one, but still can waste CPU time.

```
void acquire(struct Lock *lock) {
  while (test_and_set(&lock->held))
    yield();
}
```

(c) Assume that `struct Lock` is modified to have a field called `waitq` which list of threads waiting to acquire the lock. Explain the problem with the implementation of `acquire` on the right.

```
void acquire(struct Lock *lock) {
  while (test_and_set(&lock->held))
    thread_block(&lock->waitq);
}
```

Question 8. [20 points] A *reader/writer lock* is used to implement two different kinds of critical sections, *read critical sections* and *write critical sections* such that

- Any number of threads can be executing read critical sections at the same time as long as no writer is currently executing a write critical section, and
- A single thread can be executing a write critical section as long as no other threads are executing either a read critical section or a write critical section

Assume that the `rwlock_t` data type represents a reader/writer lock, and that it is used as follows:

```
// initialize        // read critical section     // write critical section
// r/w lock          rw_begin_r( &rwl );          rw_begin_w( &rwl );
rwlock_t rwl;        ...read shared data...       ...read/write shared data...
rw_init( &rwl );     rw_end_r( &rwl );            rw_end_w( &rwl );
```

In your implementation of reader/writer lock operations, writers should have priority over readers. If a thread is waiting to begin a write critical section, then no threads should be able to begin read critical sections until there are no threads waiting to begin write critical sections. You can use locks, semaphores, and/or condition variables (the `lock_t`, `sem_t`, and `cond_t` data types and their operations) in your implementation. See the reference section at the beginning of the exam.

(a) `rwlock_t` data type:

```
typedef struct {









} rwlock_t;
```

(b) `rw_init` function:

```
void rw_init( rwlock_t *rwl ) {









}
```

[Question 8 continues on next page.]

(c) `rw_begin_r` function:

```
void rw_begin_r( rwlock_t *rwl ) {









}
```

(d) `rw_end_r` function:

```
void rw_end_r( rwlock_t *rwl ) {









}
```

[Question 8 continues on next page.]

(e) `rw_begin_w` function:

```
void rw_begin_w( rwlock_t *rwl ) {









}
```

(f) `rw_end_w` function:

```
void rw_end_w( rwlock_t *rwl ) {









}
```

Question 9. [10 points] Consider Dijkstra's solution to the dining philosophers problem:

```
#define N 5                             /* number of philosophers */
#define LEFT (i+N-1) % N                /* i's left neighbor */
#define RIGHT (i+1) % N                 /* i's right neighbor */
enum State {THINKING, HUNGRY, EATING}; /* a philosopher's status */
enum State states[N];   /* keep track of each philosopher's status,
                           initially THINKING */
semaphore mutex = 1;    /* mutual exclusion for critical section */
semaphore phis[N];      /* semaphore for each philosopher, init to 0 */

/* In functions below, i=philosopher id, 0 to N-1 */
```

```
/* Code executed for
   one philosopher */
void philosopher(int i)
{
  while (true) {
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}
```

```
void test(int i)
{
  if (states[i] == HUNGRY &&
      states[LEFT] != EATING &&
      states[RIGHT] != EATING) {
    /* philosopher i can eat now */
    states[i] = EATING;
    /* signal i to proceed */
    phis[i].V();
  }
}
```

```
void take_forks(int i)
{
  mutex.P();
  states[i] = HUNGRY;
  test(i);
  mutex.V();
  phis[i].P();
}
```

```
void put_forks(int i)
{
  mutex.P();
  states[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  mutex.V();
}
```

[Question 9 continues on next page.]

[Question 9 continues.]

What would happen if the calls to the `test` function were removed from the `put_forks` function? If the system will operate "normally" (meaning that all threads can continue to make progress), explain why this is the case. If an undesirable behavior occurs, explain how and why.

[Extra page for answers and/or scratch work.]