# CMPU 334 Quiz 2

April 18th, 2019

Name:

**Instructions:**

This is a closed book, closed notes exam. No electronic devices, including calculators, are allowed. You have 120 minutes. There are 8 problems and 11 pages to this exam.

**Good Luck!**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(10)</td>
</tr>
<tr>
<td>2</td>
<td>(5)</td>
</tr>
<tr>
<td>3</td>
<td>(10)</td>
</tr>
<tr>
<td>4</td>
<td>(5)</td>
</tr>
<tr>
<td>5</td>
<td>(20)</td>
</tr>
<tr>
<td>6</td>
<td>(10)</td>
</tr>
<tr>
<td>7</td>
<td>(10)</td>
</tr>
<tr>
<td>8</td>
<td>(30)</td>
</tr>
<tr>
<td>Total</td>
<td>(100)</td>
</tr>
</tbody>
</table>
1. (10 points) Banker's Algorithm

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Max</th>
<th>Need</th>
<th>Total Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 7</td>
<td>7 4 7</td>
<td>10 5 7</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>3 2 5</td>
<td>1 2 5</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>3 0 2</td>
<td>9 0 4</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 5</td>
<td>0 1 4</td>
<td></td>
</tr>
</tbody>
</table>

The above table shows four processes, P0 through P3. Each processes needs a number of resources (of types A, B, and C) to complete. If a process obtains all the resources it needs, it will be able to finish and return its resources back to the system. The Allocation column shows how many of each resource is currently allocated for each process. The Max column shows the total number of each resource the process needs to be able to finish. The "Total Resources" box shows the total number of each resource the Operating System has. In other words, this is the number of resources that the OS had available before it allocated any of its resources to any process.

Part A: Using the definitions of the Banker's Algorithm we went over in class, fill in the Need column in the above diagram.

See above.

Part B: Show that the system is in a safe state by demonstrating an order in which the processes may complete.

First, write down what resources are currently available (not allocated to a process) in the system: 3 3 4
The only process that has the resources it needs to complete is P3. When P3 completes, available is: 5 4 5
Of the remaining processes, only P1 has the resources to run to completion. Available is now: 7 4 5.
Then P2 may complete and available is now 10 4 7, which is enough resources for P0 to complete. So the ordering is:

P3 -> P1 -> P2 -> P0

Part C: If a request from process P2 arrives for (1, 0, 1) can the request be granted immediately? If it can, show the system is in a safe state by demonstrating an order in which the processes may complete.

The request should not be granted. If the request is granted, Available would be: 2 3 3, and the need for P2 is 5 0 1. P3 can complete, returning its resources, increasing Available to 4 4 4. No other process can get the resources it needs to complete, so if the request is granted, deadlock may occur in the system.

Part D: If a request from process P0 arrives for (0, 2, 0) can the request be granted immediately? If it can, show the system is in a safe state by demonstrating an order in which the processes may complete. For this problem, assume that system is in the same state as it was before the request in Part B came in (i.e., the state of the system is as shown in the diagram above).

The request may be granted. A safe sequence of processes is: P3 -> P1 -> P2 -> P0.
2. (5 points) Here is an attempted solution for the producer for the producer/consumer problem. What is wrong with the following code? You can assume there are no syntax errors with the code (i.e., it compiles correctly), all function calls return without any errors, and everything has been initialized correctly.

```c
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        if (count == MAX)
            pthread_cond_wait(&empty, &mutex);
        put(i);
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
}
```

What is wrong with the above code, and how would you fix it?

The if statement should be a while loop. The correct code is shown below. When a consumer signalled that a producer slot is available, there is no guarantee that the signaled producer runs right away (mesa semantics). It is possible for a newly arriving producer to run and fill that available slot, leading to a possible buffer overrun if the signaled producer does not recheck the value of count.

```c
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == MAX)
            pthread_cond_wait(&empty, &mutex);
        put(i);
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
}
```
3. (10 points) Below is most of the implementation of a semaphore. Please help me finish it. You do not need to modify the `sem_t` struct or the `sem_init()` function, you only need to finish the implementation of `sem_wait()` and `sem_post()`. You may use the `cond_wait()` and `cond_signal()` functions (be sure to use the proper arguments when calling these functions), and can assume all function calls return normally (without any errors). You do not have to maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads.

```
// Don't need to modify
typedef struct __sem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} sem_t;

// Only one thread can call this once before use
// Don't need to modify
void sem_init(sem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    mutex_init(&s->lock);
}

// Finish this implementation
void sem_wait(sem_t *s) {
    mutex_lock(&s->lock);
    while (s->value <= 0) {
        cond_wait(&s->cond, &s->lock);
    }
    s->value--;
    mutex_unlock(&s->lock);
}

// Finish this implementation
void sem_post(sem_t *s) {
    mutex_lock(&s->lock);
    s->value++;
    cond_signal(&s->cond);
    mutex_unlock(&s->lock);
}
```

Notes on the implementation: The semaphore should block if the value is less than zero, otherwise it should return right away. In `sem_wait()`, the decrement of value *must* come after the while loop. To see why, imagine a scenario where the semaphore is initialized to 0 and two threads immediately call `sem_wait()`. If the decrement comes before the `cond_wait()`, value will be -2. When another thread calls `sem_post()`, value will be -1, and the thread will stay in the while loop and call `cond_wait()` again.

Also note that `cond_signal()` only takes the condition variable as an argument. It does not take a lock.
4. (5 points) I was working on implementation of spin-locks, but I got interrupted by pre-registration advising and was unable to finish it. Please help me finish it by completing the missing line in `lock()` and the missing line in `unlock()` (and consider taking my OS Intensive next semester). :) 

```c
// Assume this function executes atomically
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new; // store 'new' into old_ptr
    return old; // return the old value
}

typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

// Fill in the missing line below
void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}
// Fill in the missing line below
void unlock(lock_t *lock) {
    lock->flag = 0;
}
```
5. (20 points) Multiple choice

In order for deadlock to occur, which of the following conditions need to hold? Circle all that apply.

A. Threads hold resources allocated to them while waiting for additional resources.
B. There is a circular wait for resources.
C. There is a total ordering on requesting resources.
D. Threads claim exclusive control of resources they require.
E. Resources cannot be forcibly removed from the threads that are holding them.

Note: putting a total ordering on resources and having all threads attempt to acquire resources strictly in that order, will prevent deadlock from occurring.

You are trying to detect deadlock in your system. You run your deadlock detector and you find a cycle in your resource allocation graph. Which of the following are true statements? Circle all that apply.

A. If there is only one of each resource type, deadlock has occurred.
B. If there is only one of each resource type, deadlock might occur (but you are not sure).
C. If there are multiple resources of each type, deadlock has occurred.
D. If there are multiple resources of each type, deadlock might occur (but you are not sure).
E. If there are multiple resources of each type, there is no deadlock, but starvation might occur.

Under what conditions is it a good idea to use spin locks? Circle all that apply.

A. On a single cpu system with a non-preemptive scheduler due to the simplicity of the spin lock.
B. When locks are held for a long period of time.
C. When locks are held for a short period time.
D. On a system with really fast context switches.
E. To avoid priority inversion when you have different priority threads.

Note: choice A would lead to deadlock because, the scheduler would never interrupt the spinning.

Here's your page table question. At least it's not a page table lookup! Which of the following are true statements about page tables? Circle all that apply.

A. Multi-level page table lookups are faster than a single-level page table lookup.
B. Multi-level page tables, in general, take up less space than a single-level page table.
C. In addition to the Page Frame Number, a valid bit is usually stored in the Page Table Entry.
D. TLB address space identifiers (ASID) are used to avoid having to flush the TLB on a context switch.
E. A direct mapped (E=1) TLB is preferred to a fully associative TLB.

Note: A direct mapped TLB (one TLB entry per set) can lead to conflict misses. In a fully associative TLB, a TLB entry can go anywhere in the TLB.
6. (10 points) Short answers.

A. The solution to the Dining Philosophers problem avoids deadlock by eliminating which condition that is necessary for deadlock to hold?

Circular wait

B. In the Concurrent Queues designed by Michael and Scott, what was the one “trick” they used to separate the head and tail operations (to maximize concurrency).

They insert a dummy node to keep the head and tail operations separate.

C. In the approximate counter discussed in class, discuss the tradeoffs as you increase the threshold (S) in terms of performance and accuracy relative to a precise counter.

As S increases, so does performance, but accuracy decreases.

D. If multiple threads are able to enter a critical section at the same time, a race condition may occur. What is a race condition and why is it a problem?

A race condition occurs when the specific results depend on the timing execution of the code. If context switches occur at in opportune moments, we may get incorrect results. The output of the program is no longer deterministic.

E. List one advantage and one disadvantage of using a thread over a process to perform some computation.

Advantage of using a thread: It is easy for threads to share data (all threads share the same address space).

Disadvantage of using a thread: Modifications to shared data must be coordinated or a race condition may result.
7. (10 points) True / False

Circle one:

True  False  Semaphores can be used to provide mutual exclusion.

**Binary semaphores initialized to 1 behave exactly as locks.**

True  False  Condition variables by themselves can be used to provide mutual exclusion.

You need to use them with a lock, which is passed in as the second argument to **cond_wait()**.

True  False  The function **pthread_join(tid)** waits for a thread to finish running.

True  False  When updating a variable shared by multiple threads, the sequence of **Read, Modify, Write**, should happen atomically for the program to be correct.

True  False  A two phase lock is an example of a hybrid approach.

The first phase is spinning for the lock for a short period. If it is not able to acquire the lock by spinning, it then goes to sleep and waits for the lock. It is a hybrid of spin-locks and pthread locks.
8. (30 points) Synchronization three ways.

For this problem we are going to solve a synchronization task in three different ways under three different constraints. We have three threads that we want to run in a particular order, (t1, followed by t2, followed by t3). Each thread calls a worker function (t1 calls t1_work(), t2 calls t2_work(), and t3 calls t3_work()). Each thread should not call its work function until all its predecessor threads have returned from their work functions. In other words, t2_work() should not be called until t1_work() has been called and completed, and t3_work() should not be called until t1_work() and t2_work() have been called and completed. Assume that only these three threads (t1, t2, and t3) are created and the threads can be scheduled to run in any possible order. Try to solve each problem as simply and efficiently as possible under the given constraints. You can assume all function calls return normally, without any errors.

Part A: Alone in the wilderness.

For this part you are given a struct (shown below) with a single mutex and a single integer. Think of yourself Bear Grylls being dumped in the wilderness with not a lot of resources. If Bear Grylls can start a fire with nothing more than a stick and his shoelaces, you can synchronize these threads with a mutex and an integer!

For this part you are allowed to call the following functions: mutex_unlock() and mutex_lock() (be sure to use the proper arguments when calling these functions).

```c
// Do not modify
typedef struct __worker_t {
  pthread_mutex_t lock;
  int value;
} worker_t;

// Called only once before any threads
// Do not modify
tvoid worker_init(worker_t *w) {
  w->value = 0;
  mutex_init(&w->lock);
}

// Thread 1 goes first
tvoid t1(worker_t *w) {
  mutex_lock(&w->lock);
t1_work();
w->value++;
mutex_unlock(&w->lock);
}

// Thread 2 goes second
tvoid t2(worker_t *w) {
  mutex_lock(&w->lock);
  while (w->value < 1) {
    // release the lock so t1 can run
    mutex_unlock(&w->lock);
    // get the lock back before testing
    mutex_lock(&w->lock);
  }
t2_work();
w->value++;
mutex_unlock(&w->lock);
}

// Thread 3 goes third
tvoid t3(worker_t *w) {
  mutex_lock(&w->lock);
  while (w->value < 2) {
    // release the lock so others can run
    mutex_unlock(&w->lock);
    // get the lock back before testing
    mutex_lock(&w->lock);
  }
t2_work();
w->value++;
mutex_unlock(&w->lock);
}
```

Notes: The value variable keeps track of which threads have completed running. Normally, you would not hold the lock when calling the work() functions. But in this problem, since there are only three threads that run only once (and in order), it is actually more efficient to hold onto the locks for the duration of the thread execution.
### Part B: Creature Comforts

For this part you are given a struct (shown below) with a single mutex, a single integer, and a single condition variable. How can we use this new condition variable to make the synchronization task more efficient?

For this part you are allowed to call the following functions: `mutex_unlock()`, `mutex_lock()`, `cond_wait()`, `cond_signal()`, and `cond_broadcast()` (be sure to use the proper arguments when calling these functions).

```c
// Do not modify
typedef struct __worker_t {
    pthread_mutex_t lock;
    pthread_cond_t cond;
    int value;
} worker_t;

// Called only once before any threads
// Do not modify
void worker_init(worker_t *w) {
    w->value = 0;
    mutex_init(&w->lock);
    cond_init(&w->cond);
}

// Thread 1 goes first
void t1(worker_t *w) {
    mutex_lock(&w->lock);
    t1_work();
    w->value++;
    cond_broadcast(&w->cond);
    mutex_lock(&w->lock);
}

// Thread 2 goes second
void t2(worker_t *w) {
    mutex_lock(&w->lock);
    while (w->value < 1)
        cond_wait(&w->cond; &w->lock);
    t2_work();
    w->value++;
    cond_signal(&w->cond); // broadcast OK too
    mutex_lock(&w->lock);
}

// Thread 3 goes third
void t3(worker_t *w) {
    mutex_lock(&w->lock);
    while (w->value < 2)
        cond_wait(&w->cond; &w->lock);
    t3_work();
    mutex_lock(&w->lock);
}
```

Notes: Since we have only one condition variable that both t2 and t3 are waiting on, we *must* use `cond_broadcast()` in t1. This will wake up both t2 and t3 and give both of them a chance to run. If we use a `cond_signal()`, the signal might go to t3, and t2 would never wake up.
Part C: If all you have is a hammer, everything looks like a nail.

For this part you are given a struct (shown below) with two semaphores, but we are taking away your integer! Can we solve this synchronization problem with just semaphores, the swiss army knife of concurrency tools? Of course we can!

For this part you are allowed to call the following functions: sem_wait() and sem_post() (be sure to use the proper arguments when calling these functions). Also, you need to modify the worker_init() function to tell me the initial value of your semaphores.

Note: We lost our state variable value, but the semaphores have a value associated with them which can take the place of our state variable. You can see this is the simplest implementation of the three ways we have seen to synchronize these three threads. Semaphores are right tool for this type of synchronization task.