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!

1 (10)	
2 (5)	
3 (10)	
4 (5)	
5 (20)	
6 (10)	
7 (10)	
8 (30)	
Total (100)	

1. (10 points) Banker's Algorithm

Process	**Allocation** A B C	**Max** A B C	**Need** A B C	**Total Resources** A B C 10 5 7
P0	0 1 0	7 5 7		
P1	2 0 0	3 2 5		
P2	3 0 2	9 0 4		
P3	2 1 1	2 2 5		

The above table shows four processes, P0 through P3. Each process 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.

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

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.

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 the 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).

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.

```
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?

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);











    mutex_unlock(&s->lock);
}

// Finish this implementation
void sem_post(sem_t *s) {
    mutex_lock(&s->lock);











    mutex_unlock(&s->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). :)

```
// 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 (______________________________________________________________________)

        ; // spin-wait (do nothing)

}

// Fill in the missing line below
void unlock(lock_t *lock) {

    ______________________________________________________________________________;

}
```

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.

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 of time.

D. On a system with really fast context switches.

E. To avoid priority inversion when you have different priority threads.

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 identifier (ASID) are used to avoid having to flush the TLB on a context switch.

E. A direct mapped (E=1) TLB is prefered to a fully associative 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?

____

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).

____

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.

____

____

____

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?

____

____

____

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

Advantage of using a thread: ____

____

____

Disadvantage of using a thread: ____

____

____

7. (10 points) True / False

Circle one:

True False Semaphores can be used to provide mutual exclusion.

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

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.

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 as 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).

```
// Do not modify
typedef struct __worker_t {
    pthread_mutex_t lock;
    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);
}
```

```
// Thread 1 goes first
void t1(worker_t *w) {



    t1_work();



}
```

```
// Thread 2 goes second
void t2(worker_t *w) {



    t2_work();



}
```

```
// Thread 3 goes third
void t3(worker_t *w) {



    t3_work();



}
```

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).

```
// 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) {



    t1_work();



}
```

```
// Thread 2 goes second
void t2(worker_t *w) {



    t2_work();



}
```

```
// Thread 3 goes third
void t3(worker_t *w) {



    t3_work();



}
```

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.

```
// Do not modify
typedef struct __worker_t {
    sem_t s1, s2;
} worker_t;

// Called only once before any threads
// Initialize your semaphores below
void worker_init(worker_t *w) {

    sem_init(&w->s1, 0, ____________);

    sem_init(&w->s2, 0, ____________);

}
```

```
// Thread 1 goes first
void t1(worker_t *w) {









    t1_work();









}
```

```
// Thread 2 goes second
void t2(worker_t *w) {









    t2_work();









}
```

```
// Thread 3 goes third
void t3(worker_t *w) {









    t3_work();









}
```