Instructions:

This is a closed book, closed notes exam. No electronic devices, including calculators, are allowed. You have 120 minutes. There are 6 problems and 10 pages to this exam.

*Good Luck!*

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (15)</td>
<td></td>
</tr>
<tr>
<td>2 (15)</td>
<td></td>
</tr>
<tr>
<td>3 (16)</td>
<td></td>
</tr>
<tr>
<td>4 (16)</td>
<td></td>
</tr>
<tr>
<td>5 (14)</td>
<td></td>
</tr>
<tr>
<td>6 (24)</td>
<td></td>
</tr>
<tr>
<td><strong>Total (100)</strong></td>
<td></td>
</tr>
</tbody>
</table>
1. (15 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>3 5 2</td>
<td></td>
<td>6 6 8</td>
</tr>
<tr>
<td>P1</td>
<td>2 1 2</td>
<td>5 2 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>2 1 2</td>
<td>2 3 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>1 2 2</td>
<td>1 3 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 processes.

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 (0, 1, 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 P1 arrives for (1, 0, 2) 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).
2. (15 points) You run the following code shown below and you see you get the incorrect answer of \( \text{counter} = 1 \).

```c
#include <stdio.h>
#include <pthread.h>

static volatile int counter = 0;

void * mythread(void *arg) {
    int i;
    for (i = 0; i < 2; i++) {
        counter = counter + 1;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL, mythread, NULL);
    pthread_create(&p2, NULL, mythread, NULL);

    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

To investigate, you disassemble the thread and you get the following assembly code:

```
0000000000400647 <mythread>:
  400647:   b8 00 00 00 00          mov    $0x0,%eax
  40064c:   eb 12                   jmp    400660 <mythread+0x19>
  40064e:   8b 15 f8 09 20 00       mov    0x2009f8(%rip),%edx        # 60104c <counter>
  400654:   83 c2 01                add    $0x1,%edx
  400657:   89 15 ef 09 20 00       mov    %edx,0x2009ef(%rip)        # 60104c <counter>
  40065d:   83 c0 01                add    $0x1,%eax
  400660:   83 f8 01                add    $0x1,%eax
  400663:   7e e9                   jle    40064e <mythread+0x7>
  400665:   b8 00 00 00 00          mov    $0x0,%eax
  40066a:   c3                      retq
```

You noticed that when you got the incorrect answer \( \text{counter} = 2 \), You saw the following interleaving of threads.

T1 --> T2 (finished) → T1 (finished).

The first time T1 ran, how many **assembly instructions** did T1 execute before being switched to T2? _______________
The above code is an example of which of the following (circle all that apply)

A. Deterministic computation
B. Race condition
C. Mutual exclusion
D. Indeterminate program

Fix the example code below using semaphores. Assume all functions return without errors (i.e., your code does not have to check for errors).

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

// Here are the pthread semaphore functions to use:
// int sem_init(sem_t *sem, int pshared, unsigned int value); # use 0 for pshared param
// int sem_post(sem_t *sem);
// int sem_wait(sem_t *sem);

static volatile int counter = 0;

void * mythread(void *arg) {
    int i;
    for (i = 0; i < 2; i++) {
        counter = counter + 1;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL, mythread, NULL);
    pthread_create(&p2, NULL, mythread, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```
3. (16 points) Wait for me! We have seen examples of using synchronization primitives to allow one thread to wait for another. For this problem, we will use a lock and a condition variable to implement two functions, `wait()` and `work()`. When called, the `wait()` function will not return until other threads have called `work()` at least three times. In other words, the `wait()` function should wait until the `work()` function has been called three or more times first. There is also a global variable, `status`, which you may also use if you find it helpful. You may assume that there is only one thread who calls `wait()`, and you do not care which threads call `work()`, only that it has been called at least 3 times before returning.

```c
#include <stdio.h>
#include <pthread.h>

// Variables you can use in your implementation. Do not declare any other variables.
static volatile int status;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// Lock and condition variable functions you may use
// int pthread_mutex_lock(pthread_mutex_t *mutex);
// int pthread_mutex_unlock(pthread_mutex_t *mutex);
// int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
// int pthread_cond_signal(pthread_cond_t *cond);

// Called once before wait() and work() are called
void init() {
    // Note, your lock and condition variable have already been initialized (see above).
    // The only thing you need to do here is to set the initial state of status.

    status = ____________________;
}

// Return only if work() has been called at least three times. Should sleep if not able //
// to return (i.e., no busy waiting).
void wait() {

}

// Called to signify that a thread has done work
void work() {

}
```
4. (16 points) Below is a partial implementation of the producer consumer (bounded buffer) problem. Please help me finish it.

```c
int buffer[MAX];
int fill_ptr = 0, use_ptr = 0, count = 0;
cond_t empty, fill;
mutex_t mutex;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = _____________________________; // fill in the blank
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = ______________________________; // fill in the blank
    count--;
    return tmp;
}

void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == MAX) {
            // add code here
        }
        put(i);
        // add code here
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    for (int i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 0) {
            // add code here
        }
        int tmp = get();
        // add code here
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```
5. (14 points) In class, we talked about several different hardware instructions used to build locks. For this question, we'll
discuss two instructions, the load-linked instruction and the store-conditional instruction, which are used together
to implement locks in some hardware architectures, such as arm. The load-linked instruction operates much like a
typical load instruction and simply fetches a value from memory and places it in a register. The key difference comes with
the store-conditional instruction, which only succeeds (and updates the value stored at the address just
load-linked from) if no intervening store to the address has taken place. In the case of success, the
store-conditional returns 1 and updates the value at ptr to value; if it fails, the value at ptr is not updated and a 0
is returned. The C pseudo code for these instructions are shown below.

```c
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

Given the two functions above, where each function executes atomically with respect to itself, finish the implementation of
the lock using these functions.

```c
typedef struct __lock_t {
    int flag;
} lock_t;

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

void unlock(lock_t *lock) {
    lock->flag = 0;
}

// Finish the implementation of lock
void lock(lock_t *lock) {
    // Code goes here
}
```
6. (24 points) Multiple choice, fill in the blanks, and short answer.

(A) Approximate counters, a lock based concurrent data structure, trades off less ___________________________ for better ___________________________. (be as specific as you can)

(B) The following code compiles with no errors or warnings, but when you run the thread code below your return value is giving you an unexpected (i.e., wrong) answer. What is the problem with the following code and how would you fix it? You don't have to code up your fix, just explain how you would fix it.

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    int val1, val2;
    myret_t ret;
    myarg_t *m = (myarg_t *) arg;
    // some code (not shown) to compute val1 and val2
    ret.x = val1;
    ret.y = val2;
    return (void *) &ret;
}

What is wrong with the above code: ________________________________________________________________

______________________________________________________________________________________________

How do we fix it? ________________________________________________________________

______________________________________________________________________________________________

(C) Give an example of where a spinlock lock would be preferable to a lock with sleep queues and why.

______________________________________________________________________________________________

______________________________________________________________________________________________
(D) You have the following code:

```c
// This function executes atomically
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}

typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn);
        // spin
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}

Does this code guarantee mutual exclusion? Yes or No (circle one)

Does this code implement locking fairly? Yes or No (circle one)

Does this code have good performance on a single processor system? Yes or No (circle one)

(E) The solution to the Dining Philosophers problem avoids deadlock by eliminating which condition(s) that are necessary for deadlock to hold? (circle all that apply)

A. Mutual exclusion
B. Hold and wait
C. No preemption
D. Circular wait
(F) Why is it a good idea to recheck the condition that caused a thread to call `pthread_cond_wait()` when that thread is woken up with `pthread_cond_signal()`?

_________________________________________________________________________________________________
_________________________________________________________________________________________________
_________________________________________________________________________________________________

(G) For any problem you can solve with semaphores, you can also solve using condition variables (with a corresponding lock for the condition variable).

   TRUE  or  FALSE  (circle one)
   
   Explain why or why not: ________________________________________________________________
_________________________________________________________________________________________________
_________________________________________________________________________________________________
_________________________________________________________________________________________________

(H) Last question! Unless you are hopping around this quiz. In that case, keep on going, you can do it!

What was the one concept in this section of the course (concurrency) that made the most sense to you?

What was the one concept in concurrency that was the most confusing and you wished we spent more time going over?