ECE 254 F18 Final Exam Solutions

(1.1)

```
typedef struct node {
  pthread_t t;
  struct node * next;
} node;

node* head;
double total = 0.0;
pthread_mutex_lock lock;

void* query( void* arg ) {
  course_term* ct = ( course_term* ) arg;
  double aus = query_db( ct );
  pthread_mutex_lock( &lock );
  total += aus;
  pthread_mutex_unlock( &lock );
  free( ct );
  pthread_exit( NULL ); /* 0 is equivalent */
}
```

```
int main( int argc, char** argv ) {
  pthread_mutex_init( &lock, NULL );
  head = NULL;

  while( 1 ) {
    course_term* next = get_next( );
    if ( next == NULL ) {
      break;
    }
    node* n = malloc( sizeof( node ));
    n->next = head;
    head = n;
    pthread_create( &n->t, NULL, query, next );
  }

  node* n = head;
  while( n != NULL ) {
   pthread_join( n->t, NULL );
   node* old = n;
   n = n->next;
   free( old );
  }
  printf( "Total␣AUs:␣%g\n", total );

  pthread_mutex_destroy( &lock );
  return 0;
}
```

You could do some things differently, of course, such as sum up `total` using the return values of the threads rather than a lock around modification of the total. Grading notes:

- Query function casts the argument - 1
- Query function queries the database - 1
- Query function uses `pthread_exit`, either with a pointer to a double containing AUs, NULL, or 0 - 1
- Call to `get_next` is not done in parallel (either in `main` or inside a lock) - 2
- New `node` is allocated correctly - 1
- New `node` is added to list correctly - 1
- New thread is created correctly - 3
- Iteration over existing threads - 1
- Threads are joined correctly - 3
- Total is calculated without race condition (either using lock or join return value) - 3
- Total AUs is printed - 1
- All `course_term` structs are freed - 1
- All `node` structs are freed - 1

With a deduction of 1 mark for anything else allocated or initialized but not cleaned up (e.g., created a lock and used it correctly, but did not destroy).

(1.2) More Threads: The following should appear in the student's answer for full marks:

- +1: If producers are the limiting factor, adding more producers will speed things up, otherwise it will not.
- +1: If consumers are the limiting factor, adding more consumers will speed things up, otherwise it will not.
- +1: No At some point adding more threads will no longer increase the performance of the system and adding even more after that can negatively impact the performance of the system. (This is due to many different reasons ranging from hardware limitations, memory/stack limitations, operating scheduling, overhead of reading/writing to shared memory and mutual exclusion zone causing sequential access – none of these reasons are required to earn the point)

Implementation: The following should be present in the student's code for full marks:

- +1 - declare the mutexes: `in_mutex`, `out_mutex`
- +0.5 - initialize the mutexes
- +0.5 - destroy the mutexes
- +1 declare 4 semaphores: `in_items`, `in_spaces`, `out_items`, `out_spaces`
- +1 initialize the items semaphores to 0
- +1 initialize the spaces semaphores to the queue sizes
- +1 destroy all semaphores
- +2.5 Reading the work:
    1. `sem_wait( &in_items )`
    2. `pthread_mutex_lock( &in_mutex )`
    3. `item* next = pop( &in_queue )`
    4. `pthread_mutex_unlock( &in_mutex )`
    5. `sem_post( &in_spaces )`
- +1 Actually doing work in the thread: `item* worked = do_work( next )`
- +2.5 Passing worked item to next producer/consumer:
    1. `sem_wait( &out_spaces )`
    2. `pthread_mutex_lock( &out_mutex )`
    3. `push( &out_queue, worked )`
    4. `pthread_mutex_unlock( &out_mutex )`
    5. `sem_post( &out_items )`
- (-1 points) If the student uses an infinite loop to check the `in_queue.size`

(1.3) More Pizza!

One Ingredient Only: This still has a deadlock. Example: at the beginning of time, if the host puts out cheese, it gets taken by contestant A. Then the host is signalled by contestant A. The host puts out cheese again. Nobody currently is waiting for cheese so all contestants remain blocked. Deadlock. The same would occur for any other ingredient put out twice at the start of time (or any similar start point).

The Host Just Does Whatever They Want: Deadlock can still occur. At the beginning of time, if the host puts out cheese, it gets taken by contestant A. The host puts out cheese again. Nobody currently is waiting for cheese so all contestants remain blocked. The host puts out cheese a third time, but cannot put out anything else now. Deadlock. The same would occur for any other ingredient put out three times at the start of time (or any similar start point).

The Host Is Also Hungry: Deadlock will no longer occur. The host can keep placing ingredients on the table, replacing ones that are there, until an ingredient is placed that some contestant wants right now. Ingredients that contestants do not want right now get taken off the table eventually, and at some point will be replaced with ones that a contestant does want. So deadlock will not occur.

(2.1)

Either one of the following solutions is acceptable (and for the left side you could swap locks 1 and 2 and it's equivalent).

```
void do_something( ) {
  pthread_mutex_lock( &lock1 );
  int locked2 = pthread_mutex_trylock( &lock2 )
      ;
  while( locked2 != 0 ) {
    pthread_mutex_unlock( &lock1 );
    locked2 = pthread_mutex_trylock( &lock2 );
    pthread_mutex_lock( &lock1 );
  }
  /* Critical Section */
  pthread_mutex_unlock( &lock2 );
  pthread_mutex_unlock( &lock1 );
}
```

```
void do_something( ) {
  while( 1 ) {
    int locked2 = pthread_mutex_trylock( &lock2
         );
    int locked1 = pthread_mutex_trylock( &lock1
         );

    if ( locked1 != 0 && locked2 == 0 ) {
        pthread_mutex_unlock( &lock2 );
    } else if ( locked1 == 0 && locked2 != 0 )
        {
        pthread_mutex_unlock( &lock1 );
    } else if (locked1 == 0 && locked2 == 0) {
        break;
    }
  }
  /* Critical Section */
  pthread_mutex_unlock( &lock1 );
  pthread_mutex_unlock( &lock2 );
}
```

2 marks for checking the return values of trylock; 2 marks for unlocking if we were only partially successful and retrying; 1 mark for unlocking all at the end.

(2.2)

- Before the call to `foo()`, take a copy of all of the structures.
- If the watchdog timer fires, overwrite the structures with the copy of the data taken previously.
- If we've retried n times, give up and stop retrying.
- Otherwise, try calling `foo()` again.
- If `foo()` succeeds, or we give up, then we can throw away the copies of the structures.

(2.3) 0.5 mark for the answer yes or no, 1.5 for the explanation. 0 if no explanation is provided (regardless of whether the yes or no is correct).

1. No, deadlock can no longer occur. Your argument can be either (1) the pigeonhole principle: if there are 5 chopsticks and 4 philosophers, then at least 1 philosopher must have 2 chopsticks; or (2) there are insufficient requests to build a cycle (a diagram would illustrate)

2. Yes, deadlock can still occur. There is a chance that all philosophers choose the same direction at the same time. The most likely outcome is that by chance they all try to pick up the left chopstick first and all acquire the one to their left and become blocked waiting for the one to their right. The 20% chance of going the opposite way makes deadlock less likely but it could still happen.

(3.1) Virtual memory:

Part 1. Diagram should resemble:

```
-------------------------------------------
|      20 bits page #      | 12 bits offset |
-------------------------------------------
```

Part 2. We send the TLB the process and page number; it returns the frame. We keep it up to date by putting a new entry in the TLB if the CPU issued a process+page combination that did not exist in the TLB.

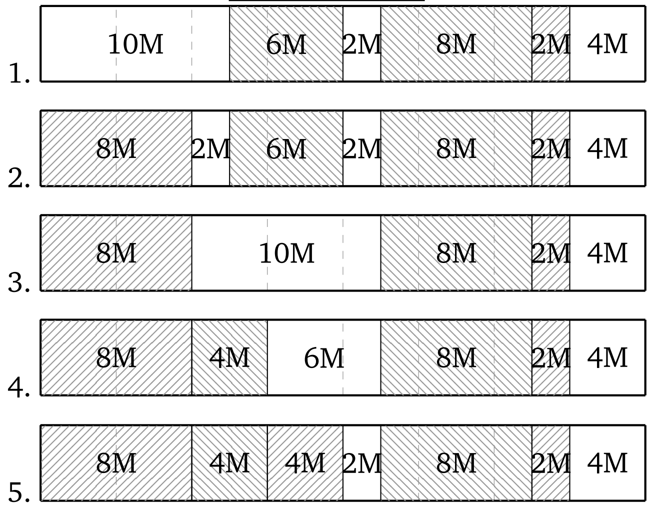
Part 3. No. The frame number is the physical location in memory; two different pages cannot be in the same physical location.

Part 4 For each of these, the least three hex digits is the offset. The rest is the page number; replace with frame number. If the frame is not present, we write page fault. For the last one, there is no page C (= 12) for process B so seg fault.

Request Table

Process	**Virtual Address**	**Result**
A	0x0000051A	0x0000551A
B	0x0000342A	0x0001F42A
A	0x00001100	Page Fault
A	0x00001BFF	Page Fault
B	0x0000CAFE	Seg Fault

Second-Best Fit



(3.3) The aging thread:

```
void* aging_thread( void* a ) {
  int * sleep_time = (int*) a;
  while( !quit ) {
    pthread_mutex_lock( &lock );
    for ( int i = 0; i < TABLE_SPACE; ++i ) {
      if ( platters[i].uses != -1 ) {
        platters[i].uses =
          platters[i].uses >> 1;
      }
    }
    pthread_mutex_unlock( &lock );
    sleep( *sleep_time );
  }
  pthread_exit( NULL );
}
```

- 1 mark for locking the mutex before making any changes
- 1 mark for unlocking the mutex BEFORE going to sleep!
- 1 mark for correctly iterating over the platters
- 1 mark for checking if the uses is -1 (right shift on negative numbers is undefined but more importantly the check for -1 is how we know if a platter is empty to don't mess with this!)
- 1 mark for performing the right shift to age and assigning it back correctly

Changes to make; 1 mark for the change, 0.5 for the location:

- at the beginning of the while loop (after line 4) lock the mutex `lock`
- At the end of the while loop (after line 38), unlock the mutex `lock`
- replace line 14 with `platters[j].uses = platters[j].uses | LEFTMOST`
- replace line 36 with `platters[rep_idx] = LEFTMOST` (or you can assign `0 | LEFTMOST`, it means the same thing).

(4.1) Each answer below is only an example; any reasonable equivalent answer would be accepted.

1. It's acceptable to allow low priority processes to starve in a system where there are consequences for high priority processes not finishing in time. This could be, for example, a real-time system where failure of a high priority process endangers the safety of people.
2. High priority threads could, for example; (1) get longer time slices, (2) be woken up first when a many threads are blocked on a mutex/semaphore; (3) get to go first when making I/O requests (e.g., disk read).
3. Yes. If CPU1 is a hyperthreading core, it shares the same hardware as CPU0 and this means that if a thread is going to be moved from CPU0 to CPU1 then there would be no cache misses (no penalty) that would occur from moving a thread from one core to another.
4. Group scheduling means that all the processes of user A can be treated as one group and all the processes of user B as another. Then, equal time is given to group A and to group B, so a user that starts 100 processes does not get 25x more CPU time than a user that starts 4.
5. A time slice every 2 ms means it happens 500 times per second. If the scheduler takes 2500 cycles each time that is 1 250 000 cycles spent scheduling in a second. 100 MHz is 1×10^8 cycles and $1\ 250\ 000/(1 \times 10^8)$ = 0.0125 = 1.25%.

(4.2) Some sample answers (any 3 of which would be fine)

- Switch the CPU to lower power mode when idle – although the idle task can be used to do useful work, when on battery, it makes sense to not do this.
- Tasks such as installing updates (app or operating system updates) can be put off until plugged in; since these things are not urgent but are effort intensive (use a lot of CPU/network/etc).
- Don't schedule background tasks/services: background tasks that would use a lot of resources can be paused by the operating system so they don't use up (as much) battery by not using as much CPU or I/O time.
- Group up I/O requests: if we can group the requests and send them together this reduces the amount of time the I/O device is active and reduces battery life. With wireless network, for example, the transmitter is in a high power state for some period of time after data is sent, so we want to minimize the occurrence of that.

(5.1)

Algorithm	**Service Order**	**Cylinders Moved**	**Improvement over FCFS**
First-Come First Serve (FCFS)	20, 302, 396, 105, 397, 280, 380	1545	1.000
LOOK	396, 397, 380, 302, 280, 105, 20	385	4.012
SSTF + Double Buffering, Buffer size 4	396, 302, 105, 20, 280, 380, 397	760	2.033

(5.2)

1. Data Transfer Mode: Character oriented - keyboard can't operate on blocks, it's one character/key combination at a time.
2. Access Method: sequential: you always get the "next" character.
3. Transfer schedule: synchronous; data always take the same amount of time to be sent/received
4. Dedication: shared; you can interrupt typing in one window and start typing in another and then go back to the first window without any problem.
5. Transfer direction: bidirectional! The little light on the keyboard that shows num lock or caps lock or anything else is technically output. And of course it is mainly for input.

(5.3) 1. Read and write for the owner (1 mark), read for the group (0.5 marks), no access for everyone (0.5 marks)

2. Make a new group "abc" (or any other name). Put the file in that group. Assign these 10 users to that group. Make the permissions on the file such that everyone has read access but the group abc has no access.