ECE 350 S23 Final Exam Solutions

(1.1)

```
void terminate_processes( ) {
  pcb_array a = get_active_pcbs( );
  for ( int i = 0; i < a.length; i++ ) {
    kill( a.ids[i], SIGTERM );
  }
  sleep( 60 );
  free( a.ids );
  a = get_active_pcbs( );
  for ( int j = 0; j < a.length; j++ ) {
    kill( a.ids[j], SIGKILL );
  }
  free( a.ids );
}
```

(1.2) The lottery system will prevent starvation if the random number generation is sufficiently random. Every thread will have some nonzero chance of winning the lottery selection each time, and given enough random draws, even a low priority thread will eventually get a turn and proceed.

(2.1) So you could argue yes or no here, but you have to justify your answer. The question is a little tricky in that it tells you about allocation, but asks about allocation and deallocation. Therefore, your answer has to say something about your assumption around how deallocation works.

If you argue yes, you have to say that it works if deallocation also has a bounded worst-case time – if you are imagining it is like `free()` where it just marks the memory as deallocated without doing any cleanup, then it would work.

If you argue no, your argument would be that the deallocation isn't guaranteed to be bounded worst-case time, either because the question doesn't tell you that or because you imagine the implementation involves coalescence or similar.

(2.2)

Add a thread to the ready queue Use the priority of the process as the index to the array of queues. Then add it to the back by adding it to the tail of the queue. If the queue has a tail pointer this can be done in constant time.

Select the next non-empty queue to dequeue from The real answer here is that when we talk about something being $O(n)$, what's the n that we're talking about? We are talking about the n here being the number of processes in the system. The number of iterations of a loop checking the bit mask is actually unrelated to the number of threads in the system. So the runtime here, being unrelated to the number of threads, is considered constant.

If your answer just says that 140 is a fixed number and so you can iterate over all of them and you have a worst case behaviour of 140 steps that's not enough for full marks because it still looks like linear behaviour with a maximum bound.

You can also argue that there exist efficient hardware instructions or similar to find the maximum bit in the bit array and given 140 priority levels is broken down to 5 32-bit words or 3 64-bit words then finding the max can easily be done in a fixed number of operations (3 or 5) and that would also be valid for full marks.

Dequeue a thread from the selected queue Given that a queue will have a pointer to the head of the list, dequeuing this element and updating the head pointer can easily be done in constant time.

Swap the Active and Expired Queues This is just swapping two pointers, so it's trivial to do this in constant time.

(2.3)

A	B	B	B	A	A	–	C	C	C	C	D	D	D	D	E	E	C	C	F

(2.4)

```
struct task {
  int          task_id;
  unsigned int priority;   /* lower values represent higher priority */
  int          ready;      /* a value of 0 means blocked (not ready);
                                    any non-zero value means ready */
  /* ... additional data ... */
};

void dispatcher (struct scheduling_queue queues[], int num_queues) {
  struct task* current = get_current_task( );
  enqueue( queues[current->priority], current );
  struct task* next  = NULL;

  for (int i = 0; i < num_queues; ++i) {
      next = dequeue_ready_task( queues[i] );
      if ( next != NULL ) {
        break;
      }
  }
  dispatch( next );
}
```

Is this vulnerable to starvation? Yes! The algorithm always takes the highest-priority ready process so a process with very low priority may never be selected to run.

(3.1)

The pointer tells us something about the number of cache misses in the system: if the pointer is moving quickly, it means there are a lot of misses so the system is probably operating slowly while it waits for data; if the pointer is moving slowly then it means the data needed is found in cache so it is likely the system is performing well.

(3.2)

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
- 0.5 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!)
- 0.5 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).

(3.3)

$0.98 * 3ns + 0.02 * (0.9999 * 300ns + 0.0001 * 8ms) = 24.9394ns$

$0.98 * 3ns + 0.02 * (0.9999 * 300ns + 0.0001 * 70us) = 9.0794ns$

$24.9394/9.0794 = 2.747\times$ speedup.

(3.4) You could argue either yes or no but I'm going to argue yes. Guard pages do increase the amount of memory each process is using, and adds additional overhead for allocating it, configuring it, and deallocating it when it's no longer needed. However, it means that programs with bugs like writing past the end of an array are much more likely to consistently encounter a segmentation fault, and therefore it's more likely the bug gets fixed in development rather than causing arbitrary bad behaviour at runtime.

(4.1)

File Allocation Method	**Allocated Blocks for File F**	**Allocated Blocks for File G**
Contiguous Allocation, First Fit	18, 19, 20, 21	2, 3
Contiguous Allocation, Worst Fit	18, 19, 20, 21	22, 23

(4.2)

Part 1: To find bad blocks, you would just want to write some known bit pattern to each block and try to read it back. If you're able to do so successfully then chances are the block is fine; if what you read back is not the same as what you wrote then you know the block is bad.

That's a basic version. You can come up with a more detailed one where you try several passes and write different known patterns to try to identify problems. The basic version is enough to get the marks here, though.

Part 2: If a block has previously been identified as bad, the operating system can mark it as "bad", that is a distinct state other than free/allocated. Or you could mark it as allocated to some dummy or system file. Either way, this would prevent it from being selected as a free block when a new one is needed for an allocation.

(4.3)

Add item to buffer:

```
lock mutex
if current size + item size > capacity
  trigger empty
  current size = 0
add item to buffer
buffer size += item size
if buffer is full
  trigger empty
unlock mutex
```

Flush Buffer

```
lock mutex
if current size is not 0
  trigger empty
  current size = 0
unlock mutex
```

(4.4) Honestly, you could argue for any of these answers. As long as you have an explanation for your decision that references some principles (security, user choice, user experience, etc). That's fine!

(5.1) You don't have to choose these exact ones, as long as your answers make sense:

1. Full Backup
   (a) Pros: Easiest to implement, every backup can restore the system (most redundancy).
   (b) Cons: Slowest, Uses the most space
2. Incremental Backup
   (a) Pros: Uses much less space, faster to carry out than full backup.
   (b) Cons: Lots of work to determine which files have changed; full restore requires a lot of work to find latest versions of each file.
3. Diff-Based
   (a) Pros: Uses least space, copying diffs is faster than coping whole files.
   (b) Cons: Computationally intensive, risk of failing to apply a diff due to incompatible changes.

(5.2) Resilience is about how the system can continue to function even in the face of errors or hardware failures or similar, so the answer should talk about how failing to enforce file permissions causes the system to stop working.

Lack of enforcement could mean that a process accidentally or intentionally deletes or modifies a file that belongs to another user which would interfere with that user's process in a way that could stop it from working. The same applies to the operating system: if a change is made to a key operating system file, the whole system might fail.

(5.3) Remember that the goal is to stop errors before the system is in-use so anything that helps prevent a bug from being shipped is fault prevention.

Code Review: Code reviews allow other developers an opportunity to check the code for quality and to spot potential issues before they are created. Having one or more fresh looks at the code may reveal something new.

Unit Tests: The purpose of unit tests are to verify that the code works as expected and to fail if there is a change introduced that changes the behaviour of the code.

Design Documents: The purpose of the design document is to make key decisions up-front and have the opportunity to review them, at a time when making necessary changes is least expensive.

(5.4)

You need to have two (different) equations if you need to solve for two unknowns. Loss of a drive means losing one of the terms in the equation and loss of two drives means having two terms missing. If you just use the P calculation twice, then you really only have one equation with two unknowns and that's not enough to recover the data. Whereas the second value Q means we can rearrange these two equations to solve for both unknown variables.

(6.1) There are two major errors here. If anyone can think of others, as long as they are valid we can accept it.

1: Mike changed the header of a function in `k_mem.c`, which is not permitted.

2: Mike is returning the wrong address. This is the low address of the stack, but the stack should be given as the high address.

(6.2) There are nine errors in the code, any eight of which the student needs to find. The order does not matter for the solution. There are a couple of missing semicolons, but the goal is to find semantic errors so those aren't the things we are looking for – the question says the code compiles so the semicolons should not be an issue.

1. ERROR: `tsk_create` attempts to create a task with stack size of `0x100`, which is too small given the minimum.

FIX: stack size should be increased to at least `0x200`

2. ERROR: calling scheduler does not yield the task

FIX: task should yield instead

3. ERROR: `utask1` cannot free `utask2`'s memory

FIX: several options are available. One is to just ignore it in `utask1`. The best is to dealloc in `utask2`. However, a student may argue that we have no idea whether `utask2` even runs before `utask1` happens, since we do not know what else the kernel is doing.

4. ERROR: `utask1` is calling `k_*` function `k_tsk_exit()`

FIX: change it to `tsk_exit()` system call

5. ERROR: `utask2` does not exit properly

FIX: `utask2` should call `tsk_exit` system call

6. ERROR: `ktask1` calls `printf`. Kernel tasks are prohibited from calling stdlib functions

FIX: change the call to `SER_PutStr`

7. ERRROR: (subtle) - there is technically nothing wrong with a task setting its own state, however, the call to `k_tsk_yield` will fail now, since a prerequisite is that the `gp_current_task->state` variable is set to RUNNING for yield to succeed.

FIX: simplest fix is to just not do this and only call yield. However, a student can successfully argue that setting a task state to `DORMANT` means that we wanted to exit, so calling `k_tsk_exit` is acceptable as well

8. ERROR: kernel task calls a system call `tsk_yield`

FIX: kernel task should call `k_tsk_yield`

9. ERROR: `p1` is never deallocated

FIX: deallocate it