Please print in pen:

Waterloo Student ID Number:							

WatIAM/Quest Login Userid:							

UNIVERSITY OF WATERLOO

Examination Final Spring 2023 ECE 350

Times: Monday 2023-08-14 at 16:00 to 18:30 (4 to 6:30PM)
Duration: 2 hours 30 minutes (150 minutes)
Exam ID: 5322874
Sections: ECE 350 LEC 001
Instructors: Jeff Zarnett

Special Materials

Candidates may bring only the listed aids.
· Calculator - Non-Programmable

Instructions:

1. No aids are permitted except non-programmable calculators with no persistent memory.
2. Turn off all communication devices. Communication devices must be stored with your personal items for the duration of the exam. Taking a communication device to a washroom break during this examination is not allowed and will be considered an academic offence.
3. Place all bags at the front or side of the examination room, or beneath your table, so they are inaccessible.
4. There are six (6) questions, some with multiple parts. Not all are equally difficult.
5. The exam lasts 150 minutes and there are 120 marks.
6. Verify that your name and student ID number is on the cover page and that your examination code appears on the bottom of each page of the examination booklet.
7. If you feel like you need to ask a question, know that the most likely answer is “Read the Question”. No questions are permitted. If you find that a question requires clarification, proceed by clearly stating any reasonable assumptions necessary to complete the question. If your assumptions are reasonable, they will be taken into account during grading.
8. After reading and understanding the instructions, sign your name in the space provided below.

Signature

1 Process Management [10 Marks Total]

1.1 Sunset [7 marks]

When a shutdown is called, processes need to be informed and ideally offered an opportunity to shut down in an orderly manner. One approach for this is to use `SIGTERM` as the way of politely asking a process to shut down before a less polite `SIGKILL` is used. Write an implementation below, in C, of some logic to call the shutdown that sents `SIGTERM` to all active processes, waits 60 seconds, before sending `SIGKILL`.

You can get a list of the currently-existing processes with the function shown below:

```
typedef struct {
  pid_t* ids; /* Array of process ids */
  int length; /* Length of ids array */
} pcb_array;

/* Returns pcb_array -- deallocation of the internal array in side the struct is responsibility of the caller */
pcb_array get_active_pcbs( );
```

Remember that you can send a signal to a process with the function:

```
int kill(pid_t pid, int signal_number);
```

and that you can put the current thread to sleep with:

```
unsigned int sleep(unsigned int seconds);
```

Complete the code below.

```
void terminate_processes( ) {









}
```

1.2 Unblocking, Again [3 marks]

A colleague has another proposal for how unblocking threads should work and it is based on the lottery system of scheduling. In this case, every thread blocked on each semaphore gets some number of tickets proportional to its priority. Will this system prevent starvation? Justify your answer.

2 Scheduling [30 Marks Total]

2.1 Task Estimation [2 marks]

In real-time scheduling, we care about worst-case execution times and we discussed the idea of code analysis as a mechanism for estimating the worst-case execution time for some code. Use of `malloc()` and `free()` was on the list of things that we cannot use because these functions could take an arbitrary amount of time to run depending on how memory is managed. Suppose, however, that you have a system where the implementation for memory allocation with `malloc()` has a bounded worst-case time. Does this mean that we can now assign worst-case execution times to code that does dynamic memory allocation and deallocation? Justify your answer.

2.2 O(1) (Constant Time) Scheduler [8 marks]

In lecture, we discussed the operation of the (now-replaced) O(1) scheduler and how it improves on the O(n) runtime behaviour of the original UNIX scheduling algorithm. Explain how each of the following operations can be completed in constant time.

Add a thread to the ready queue [2 marks]

Select the next non-empty queue to dequeue from [2 marks]

Dequeue a thread from the selected queue [2 marks]

Swap the Active and Expired Queues [2 marks]

2.3 Real-Time Uniprocessor Scheduling [10 marks]

You have a real time system where all tasks are hard real time, and the scheduling algorithm is earliest deadline first, and time slicing is used. At the end of each time slice, the scheduler runs, evaluates the current state, and decides what task to run next. Assume no tasks will ever get blocked for any reason.

The table below gives the breakdown of the tasks. A task that arrives during time slice n may be scheduled during time slice $n + 1$ (but not sooner). A task that has an execution length of k requires k time slices to complete. A task with a deadline of D must complete during or before time slice D.

Task ID	Arrival Time Slice	Execution Length (time slices)	Deadline Time Slice
A	0	3	8
B	1	3	7
C	7	6	19
D	11	4	15
E	12	2	18
F	18	1	20

Each block represents a time slice. In each box below, write the ID of the task that will be executed in that time slice. If there is nothing executing in a time slice, write a dash (–) in the box.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

2.4 Scheduling Implementation [10 marks]

Suppose that the scheduling system that we will use has one queue for each priority level. When the dispatcher runs, it should take the current task and put it into the queue corresponding to its priority. The scheduling algorithm works on a FCFS basis; take the highest-priority ready task from the ready queues. When the task is selected, actually dispatch it and it will begin executing. You may assume there is an idle task so you do not have to worry about the possibility that no ready task is found.

Part i. [8 marks] Complete the code below to implement the described behaviour.

```
struct task {
  int          task_id;
  unsigned int priority;   /* lower values represent higher priority */
  /* ... additional data ... */
};

/* Return a pointer to the currently-executing task structure */
struct task* get_current_task( );

/* Dispatch the selected task */
void dispatch( struct task* task );

/* Put task t into the scheduling queue q. The task should be put into the queue of the correct level */
void enqueue( struct scheduling_queue q, struct task* t );

/* Dequeue a ready task from the queue q if any; returns NULL if this queue is empty */
struct task* dequeue_ready_task( struct scheduling_queue q );

void dispatcher (struct scheduling_queue queues[], int num_queues) {










}
```

Part ii. [2 marks] Is the scheduling algorithm as described vulnerable to starvation? Justify your answer.

3 Memory [22 Marks Total]

3.1 Cache Replacement Algorithm [4 pages]

If your system uses the "clock" (second chance) algorithm for page replacement, the term "pointer" is used to refer to the page that the hand of the clock is pointing to. If you observed the behaviour of the movement of the pointer, what could you say about the system? In your answer you should say what it means if the pointer is moving quickly and what it means if it's moving slowly.

3.2 A Feast for Dwarves [10 marks]

Recall from the in-class exercise the feast that Bilbo was hosting for the dwarves. Bilbo used the "least frequently used" (LFU) replacement algorithm, taking away dishes that were unpopular and replacing them with ones that had been requested (and dishes never ran out). This is a reasonable plan, except it doesn't account for one thing: a dish that was popular in the beginning might accumulate a high count but then not be used again for a long time. So an appetizer (such as a salad) could take up table space throughout the whole meal. The obvious solution to this is aging: if a dish has not been requested in a while, it is a good candidate for replacement.

Below on the left is an implementation of LFU replacement without aging. Actually aging the counters will be done by a different thread that you will write. Below the code, you will write a list of changes that you would make to the sample code. Clearly indicate what you would change and what line(s) would be affected or where your new code would be inserted. Concurrency control is needed, so a `pthread_mutex_t` global variable called `lock` has been created for you and you may assume it has been properly initialized and will be properly destroyed before the program exits.

To age a counter, do a bitwise shift right such as `x = x >> 1;`. To set the leftmost (valid) bit in an integer, use the bitwise OR operator, such as `y = y | LEFTMOST;` (assume that `LEFTMOST` is a constant defined for you and is correct for the size of the `int` type in this system). [Continued on next page...]

```
 1  int serve_dishes( ) {
 2    int fetches = 0;
 3    int rep_idx = 0;
 4    while( 1 ) {
 5      int next = get_next_dish_order();
 6      if ( next == -1 ) {
 7        quit = true;
 8        break;
 9      }
10      /* Check if it's already in there */
11      bool found = false;
12      for( int j = 0; j < TABLE_SPACE; ++j ) {
13        if (platters[j].dish == next) {
14          found = true;
15          platters[j].uses++;
16          break;
17        }
18      }
19      if (!found) {
20        ++fetches;
21        for( int k = 0; k < TABLE_SPACE; ++k ) {
22          /* Empty platter has uses of -1 */
23          if ( platters[k].uses == -1 ) {
24            rep_idx = k;
25            found = true;
26            break;
27          }
28        }
29        if (!found) {
30          for ( int k = 0; k < TABLE_SPACE; ++k ) {
31            if (platters[k].uses < platters[rep_idx].uses ) {
32              rep_idx = k;
33            }
34          }
35        }
36        platters[rep_idx].dish = next;
37        platters[rep_idx].uses = 0;
38      }
39      print_state( );
40    }
41    return fetches;
42  }
```

Complete the aging thread (4 marks):

```
void* aging_thread( void* a ) {
  int * sleep_time = (int*) a;
  while( !quit ) {



    sleep( *sleep_time );


  }
  pthread_exit( NULL );
}
```

Write your changes here (6 marks):

3.3 Use Budget Wisely [4 marks]

We discussed in lecture that upgrading from a magnetic hard drive to a solid state drive is a huge improvement for the system. But how much? Let's calculate the speedup using the effective access formula. Speedup S is calculated as $S = \frac{T_{old}}{T_{new}}$.

We will again use the formula: Effective Access Time $= h \times t_c + (1-h)(p \times t_m + (1-p) \times t_d)$. In your calculations, the cache hit rate is 98% and the memory hit rate is 99.99%. Cache access times are 3 ns and memory access times 300 ns. The magnetic drive has an access time of 8 ms and the SSD an access time of 70 μs.

3.4 Safety Has Its Cost [4 marks]

One possible extension of memory allocation that can be used to increase security is the idea of guard pages. A guard page is effectively an extra area of memory allocated after the requested allocation and that memory is flagged as no access, so any attempt to read or write the memory results immediately in a segmentation fault. Are these a good idea? Justify your answer, looking at both pros and cons.

4 I/O Devices and File Systems [22 Marks Total]

4.1 File Allocation [4 marks]

A very small USB drive is formatted such that it has 32 blocks numbered 0 through 31. The table below lists the files currently allocated on that drive and their respective block(s).

File ID	Allocated Block(s)
A	31, 32
B	0, 1
C	26, 27, 28
D	11, 12, 13, 14, 15, 16, 17
E	5, 6, 7, 8, 9

Then two more files are allocated: first file F with a size of 4 blocks, then file G with a size of 2 blocks. Complete the table below to show where the blocks of files F and G would be allocated, according to the algorithm listed in the leftmost column.

File Allocation Method	Allocated Blocks for File F	Allocated Blocks for File G
Contiguous Allocation, First Fit		
Contiguous Allocation, Worst Fit		

4.2 Bad Blocks, Bad Blocks, What'cha Gonna Do? [5 marks]

Hard drives can sometimes develop "bad blocks", which are parts of the drive that are damaged so that reads and/or writes do not succeed (e.g., the data is unchanged, or incorrect data is read/written).

Part 1. New Drive, Who Dis? [3 marks] It is likely that a newly-manufactured drive has some nonzero number of defective (bad) blocks from the beginning. How could you test an empty hard drive to identify the bad blocks?

Part 2. Stay Out. [2 marks] It's also possible that the drive develops bad blocks over time for various reasons. If the OS identifies that a block is bad at a later time (let's ignore how), how can the OS be sure that this block will not be used when a file is created or otherwise needs a new block?

4.3 Buffering [10 marks]

In lecture, we discussed the idea of buffering as a way of handling a mismatch between devices of different speed. Imagine you have a buffer of capacity `CAPACITY` where data will accumulate until the buffer is full, and will then be written to disk. Items may be of variable size, but are never larger than the capacity of the buffer. Multiple callers may wish to add to the buffer at a time so race conditions should not be permitted. If the buffer is full, someone wanting to add more data to the buffer should get blocked until the write is complete.

When we're finished, we typically want to flush the buffer – write everything out that's in the buffer, even if it's not full. If it's empty, there's nothing to do. Below, write down the pseudocode describing the workflows for both adding an item to the buffer and flushing the buffer. Your pseudocode is responsible for managing the state of the buffer (how full it is).

Add Item To Buffer

Flush Buffer

4.4 Vulnerable Device Drivers [4 marks]

In lecture, we discussed the problem of vulnerable device drivers being an easy attack vector for malicious actors. Suppose you work at an operating system vendor company. While your company an revoke approval for the driver with the problem and prevent its installation so that people get an up-to-date version, in the case you're considering now, the device vendor has gone out of business and will not be releasing a patch for it. [Continued on next page...]

You're an engineering manager now so the problem has landed at your desk. One of your team members, Leslie, thinks that the correct approach to this is simply to create an OS update that uninstalls the device driver in question. Another team member, Aang, thinks that the correct approach is for your company to do the work of writing a patch for the driver to fix the problem, even though you don't have the source code. A third team member, Tuli, thinks that just revoking the approval of this driver is sufficient and it's the user's call if they want to take the risk or not.

What decision do you make and why?

5 Reliability [20 marks]

5.1 What About Second Backups? [6 marks]

I'm sure you take backups of your important data, but how to take the backups matters. Let us consider three options. In all cases, the first backup is a full backup of all files, but then after that there are choices.

1. Full backup: copy the entire contents of the drive from the source to the destination, whether files have changed or not.
2. Incremental backup: copy only the files that have changed from the source to the destination.
3. Diff-based incremental backup: identify which files have changed, and copy only the diff of the change to the destination.

List two (2) pros and two (2) cons of each of these approaches.

5.2 Fault Tolerance [4 marks]

Explain how file permissions (e.g., the UNIX-style Read/Write/Execute permissions for Owner/Group/Everyone) are a useful tool for achieving fault tolerance. Your answer should discuss how a lack of enforcement of these policies may impact both important user-level processes and the operating system itself.

5.3 Fault Prevention [6 marks]

As the saying goes, prevention is better than cure. Explain how each of the following software development strategies could play a role in fault prevention for a real-time system. 2 marks each.

Code Review.

Unit Tests.

Design Documents.

5.4 RAID: Parity [4 marks]

RAID systems use a parity approach to recover data. For the sake of this question, we'll treat all data as integers and not consider bits/bytes/blocks. If our data elements are a, b, c, d, e then a simple approach to parity is to calculate $P = a + b + c + d + e$. After that, if one drive dies, it's trivial to show that no matter which value is the unknown in the equation, we can simply re-arrange the equation and solve for the missing element.

In a RAID 5 arrangement where we can suffer the loss of one drive, for example, we calculate the P value as above. In a RAID 6 arrangement where we can suffer the lost of two drives, a second parity calculation is done. But instead of just a second copy of the original parity calculation, the second parity is calculated using $Q = 1a+2b+3c+4d+5e$. This is more computationally difficult. Explain why the second parity calculation follows such a formula.

6 RTX (Lab) [15 marks total]

6.1 Stack Management [3 marks]

Your colleague, Mike, has just realized that user stacks are supposed to be dynamically allocated, not statically allocated! In a panic, Mike committed the following updated function to the main branch, which got in just before the deadline. Mike didn't test it, but hey, the group got 100% on lab 1 so he's confident that it's okay. Mike is very wrong. Why?

```
U32* k_alloc_p_stack(size_t stack_size) {
    return k_mem_alloc(stack_size);
}
```

6.2 Error Finding [12 marks]

The following code snippet is included in a project. We assure you that this code compiles, but there are many errors. You need to find and explain how to fix eight (8) errors, although the code contains more than 8 errors. For each error you identify, briefly explain what it is (0.5 marks each) and suggest a fix (1 mark each). Note: do not rewrite the entire code snippet. Instead, complete the table below to explain each error and its solution.

```
 1  U32* p1;
 2  U8* p2;
 3
 4  //This has been set up as a user task
 5  void utask1(void) {
 6      p1 = (U32*)k_mem_alloc(sizeof(U32))
 7      task_t tid;
 8      RTX_TASK_INFO task_info;
 9
10      SER_PutStr (0,"utask1:␣entering␣\n\r");
11      tsk_create(&tid, &utask2, 175, 0x100);
12
13      /* Allow task 2 to run - call the scheduler
14          to get a new task */
15      scheduler();
16
17      /* terminating - remember to free all of the
18          child task's memory or you get a leak!*/
19      k_mem_dealloc(p2);
20      k_tsk_exit();
21  }
22
23  //This has been set up as a user task
24  void utask2(void) {
25      p2 = (U8*)k_mem_alloc(sizeof(U8))
26      SER_PutStr (0,"utask2:␣entering␣\n\r");
27      /* do something */
28      long int x = 0;
29      int ret_val = 10;
30      int i = 0;
31      int j = 0;
32  }
33
34  /* This has been set up as a kernel task at boot time,
35     with priority 244 */
36  void ktask1(void) {
37      printf("In␣the␣kernel␣task!")
38      while(1) {
39          /* Yielding. Set this task's state to DORMANT
40              so the scheduler doesn't choose it */
41          gp_current_task->state = DORMANT;
42          tsk_yield();
43      }
44  }
```

#	Line(s)	Description of Problem	Description of Fix
1			
2			
3			
4			
5			
6			
7			
8			