Please print in pen:
Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination
Final
Winter 2023
SE 350

Times: Saturday 2023-04-15 at 16:00 to 18:30 (4 to 6:30PM)
Duration: 2 hours 30 minutes (150 minutes)
Exam ID: 5142618
Sections: SE 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 110 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. An API reference document is provided to accompany this exam. This will not be collected, so do not use it as extra space.
9. After reading and understanding the instructions, sign your name in the space provided below.

Signature

Please initial:

1 RTX [15 Marks]

Consider the following code that resembles the labs:

```
void proc1(void)
{
  int  i = 0;
  int  j = 0;
  void  *p_blk;
  MSG_BUF *p_msg;
  char  *ptr;
  void  *data_blk;
  data_blk = request_memory_block();
  uint32_t dataAddr = (uint32_t)data_blk;
  //Call a function to fill the block with the string
       literal provided.
  // Assume it is written above.
  fill_block_with_string((char*)data_blk,"Hello, P2");
  p_blk = request_memory_block();
  p_msg = p_blk;
  p_msg->mtype = DEFAULT;
  ptr = p_msg->mtext;
  *ptr++ = (char)dataAddr;
  *ptr++ = (char)(dataAddr >> 8);
  *ptr++ = (char)(dataAddr >> 16);
  *ptr++ = (char)(dataAdddr >> 32);
  send_message(PID_P2, p_blk);
  set_process_priority(PID_P1,LOW);
  while (1) {
   release_processor();
  }
}
```

```
void proc2(void)
{
  int  i = 0;
  int  j = 0;
  MSG_BUF *p_msg;
  void  *p_blk;
  p_blk = receive_message(NULL);
  p_msg = p_blk;
  char* data = (char*)(*(uint32_t*)p_msg->mtext);
  uart1_put_string("proc2: got a message - ");
  uart1_put_string(data);
  release_memory_block(p_blk);
  release_memory(data);
  set_process_priority(PID_P2, LOW);
  while(1) {
   release_processor();
  }
}
```

Part i. List at least two issues that prevent this code from meeting the specification of the RTX as per the lab manual.

Part ii. What is the intended operation of these two processes?

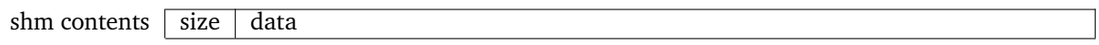
Part iii. Re-write the processes so that the intended operation is as similar as possible but the code conforms to the API requirements

Please initial:

2 Processes and IPC [13 marks]

Checksums are used to verify file integrity (i.e., make sure they haven't been corrupted). Your program will calculate the checksum of a file twice: once in a child process and again in the parent process. You will then compare the results to make sure both have the same value. The processes will communicate using shared memory.

The parent process will create a shared memory segment. The child process will read the file specified by `argv[1]` and put the file size followed by the file contents into shared memory (see diagram below).



Files will be no bigger than `MAX_FILE_BYTES`. The child process will then use `calculate()` to calculate the checksum and return that value. The parent process will use `wait()` to wait for the child process and get its return value. When the child process ends, its `return` statement adds other information to the return value, so the parent process needs to use `WEXITSTATUS(return_status)` to extract the returned checksum (it produces an unsigned integer value). The parent process will then recalculate the checksum from the data in shared memory. It returns 0 if both checksums are the same and -1 if they are not.

The file should be closed and shared memory released before returning. Complete the code below to implement the behaviour described above.

```
uint8_t calculate(void *mem, size_t bytes);

int main(int argc, char **argv) {
  uint8_t childsum;
  uint8_t checksum;

  int shmid = shmget(IPC_PRIVATE, sizeof(size_t) + MAX_FILE_BYTES, IPC_CREAT | 0600);

  int pid = fork();
  if (pid < 0) {
      printf("Error␣occurred:␣pid␣=␣%d.\n", pid);
      return -2;
  } else if (pid == 0) {









  } else {









    return childsum == checksum ? 0 : -1;
 }
}
```


Please initial:

3 Deadlock [20 marks total]

3.1 Design To Avoid Problems [10 marks]

Although we can't make deadlock categorically impossible, when we have control of the source code of a program, we can make some modifications to our program to make it so that deadlock cannot happen in that program. For each of the strategies below, name which of the four essential elements of deadlock is eliminated, and explain why it works. 2 marks each.

1. Use one mutex to control access to shared data and no other synchronization constructs exist in the program.

2. Always use `trylock` or `trywait`.

3. A thread may hold only one mutex or semaphore at a time.

4. Use copying to make it so no data is shared.

5. Strict lock ordering is observed.

3.2 Stay Out of Trouble [6 marks]

Deadlock prevention routines do not have to be perfect – and we've discussed in the lecture why it's very difficult to make one that is – but perfection is not required. Suppose that a colleague comes to you with an implementation of a deadlock prevention algorithm that is supposed to work by managing when threads are selected to run. How would you evaluate this algorithm? Your answer should include (1) a brief overview of two tradeoffs in implementing such an algorithm [2 marks]; (2) what test scenarios you would use and why [2 marks]; and (3) what things you would measure [2 marks].

Please initial:

3.3 Rollback Recovery [4 marks]

One strategy we talked about for deadlock recovery was based around rollback: the idea of setting things back in time to an earlier stage and continuing on. However, rollback does not always work because there exists the possibility that repeating a section of code means some action will happen twice and you may not want that. Give an example of the kind of action that would make it impractical to use rollback on a section of code, and explain what you could change to address that problem.

4 Memory [21 marks total]

4.1 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.

4.2 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.

4.3 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.

4.4 Virtual Memory [9 marks]

We have a virtual memory scheme that uses segmentation combined with paging. It has a physical address space of 32 bits and a virtual address space of 40 bits. Page size is 4 KB, and we want a maximum segment size of 16 MB.

Part 1. Show and explain the layout of a virtual address — that is, the breakdown of the 40 bits into segment and/or page numbers, offsets, etc.

Part 2. For this virtual memory scheme, at some point in time, we have the following segment table and page tables for the process currently executing:

Segment Information

Number	Length	Base Address
0	0x3000	0x1000
1	0x28F0	0x5000

Page Table for Segment 0

Page #	Present	Frame #
0	0	–
1	1	9
2	1	7
3	1	4

Page Table for Segment 1

Page #	Present	Frame #
0	0	–
1	0	–
2	1	3
3	1	5

Assuming both page tables are in memory, complete the table below. To do so, determine the physical address corresponding to the virtual addresses, and indicate whether the given virtual address causes a segmentation fault or a page fault (write Yes or No in the relevant box).

Virtual Address	Physical Address	Seg. Fault?	Page Fault?
0x00000030D1			
0x00000001AC			
0x0001005B18			
0x00010007FF			

5 Scheduling [21 marks total]

5.1 Unequal Round-Robin [9 marks]

Our system currently uses a round-robin (RR) scheduler, but you are considering an alternative called Unequal Round-Robin (URR). In the URR scheme, a task that is of higher priority (lower number) gets longer time slices. If a process is of priority 1, when scheduled, it can run for up to 3 times units; a process with priority 2 can run for up to 2 time units; a process with priority 3 can run for up to 1 time unit. Show the execution order (in the form $A \rightarrow B \rightarrow C...$) for the below processes (none of which will get blocked) under Round-Robin and the Unequal Round-Robin Scheme [4 marks]; also complete the table below of the completion times [2 marks].

Round Robin:

Unequal Round Robin:

Process	Total Execution Time	Priority	Completion Time RR	Completion Time URR
1	4	3		
2	7	2		
3	10	3		
4	1	1		
5	3	1		

Based on the table, calculate the improvement, if any, in average completion time [1 mark]:

The execution order assumes that there is effectively no cost to a context-switch. Imagine instead that the process of the context switch takes 0.1 time slices. How does that change, if at all, the total completion time for each approach? [2 marks]

Please initial:

5.2 Scheduling Implementation [12 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. [9 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. [3 marks] Is the scheduling algorithm as described vulnerable to starvation? Justify your answer.

6 I/O [20 marks total]

6.1 Buffering [13 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. [Continued on next page...]

Please initial:

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

6.2 Disk Scheduling [7 marks]

Consider a disk that has 500 cylinders labelled 0 through 499. Suppose the incoming disk reads waiting to be serviced are: `365, 49, 350, 144, 434`. The starting position of the disk read head is 467 and it is moving in the ascending direction. Complete the table below. Improvement is measured as FCFS cylinders moved divided by the alternative strategy's cylinders moved. When reporting the improvement over FCFS, round to 3 decimal places.

Algorithm	**Service Order**	**Cylinders Moved**	**Improvement over FCFS**
First-Come First Serve (FCFS)	365, 49, 350, 144, 434		1.000
Shortest Seek Time First (SSTF)			
SSTF + Double Buffering, Buffer size 3			
SSTF + Double Buffering, Buffer size 5			
SCAN			

Extra Space. Remember to write down what question you are answering!

Please initial: