Please print in pen:

Waterloo Student ID Number:

WatIAM/Quest Login UserID:

UNIVERSITY OF WATERLOO

Final Exam - Winter 2026 - SE 350

1. Before you begin, make certain that you have one **2-sided booklet with 10 pages**. You have **150 minutes** to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points for that question.

2. Please read all questions before starting the exam as some of the questions are substantially more time consuming. Read each question carefully. Make your answers as concise as possible. **If there is something in a question that you believe is open to interpretation, then please write your interpretation and assumptions!**

3. All solutions must be placed in this booklet. If you need more space to complete an answer, you may be writing too much. However, if you need extra space, use the blank space on the last page of the exam clearly labeling the question and indicate that you have done so in the original question.

Good Luck!

Question	Points Assigned	Points Obtained
1	**30**	
2	**24**	
3	**22**	
4	**8**	
5	**16**	
Total	**100**	

Q1. (30 points) True-False with explanation.

For each question:

- Circle your answer and write your explanation below each question.
- Explanations should not exceed 3 sentences.
- One point for correct true-false.
- One point for correct explanation.
- No points for any explanation if true-false is incorrect.

1. Adding a TLB will always improve memory access latency.

True False

- False. Any workload that doesn't visit the same page twice within the TLB's capacity will make the workload perform worse than not having a TLB.

2. Segmentation does not change the value of the bits in the offset field of the virtual address when translating to a physical address.

True False

- False. With segmentation we add base to virtual address, which could modify the offset.

3. Two distinct page tables can have the same page table entry despite existing in separate processes.

True False

- True. This happens when processes share pages with each other.

4. After the kernel handles a user page fault, the user application resumes execution at the instruction immediately following the instruction that caused the page fault.

True False

- False. it resumes by replaying the instruction that caused the page fault error so that the instruction executes successfully.

5. You can implement the MIN replacement policy in a real operating system.

True False

- False. The MIN replacement policy requires knowledge of which page will be requested furthest into the future. This is not possible in a real operating system.

6. In a system with hold-and-wait, no preemption, and mutual exclusion, a cycle in the resource allocation graph means deadlock has happened if all resources are single-unite resources.

True False

- True. In a system with hold-and-wait, no preemption, and mutual exclusion, a cycle in the resource allocation graph means deadlock has happened if all resources are single-unite resources.

7. Threads within the same process can share data (since they share the same memory space), but threads in different processes cannot share data.

True False

- False. There are many inter-process communication (IPC) methods that can be used for threads in different processes to share data.

8. With copy-on-write, we take advantage of demand paging to make a copy of all pages of the parent process for the forked process.

True False

- False. we take advantage of demand paging to copy pages out only when necessary (on the first write).

9. A thread cannot be preempted during a critical section (i.e. after successfully acquiring a lock and before releasing it).

True False

- False. The thread can still get preempted. If another thread attempts to acquire the lock that the original thread acquired, it will get blocked waiting for the lock to become free again.

10. A thread can modify the stack variables of another thread in the same process.

True False

- True. Since the threads in the same process share an address space, they can all access the same memory (including each other's stacks).

11. Round Robin scheduling always results in a lower average response time than First Come First Serve scheduling.

True False

- False. If jobs happen to arrive in sorted order, smallest burst time first, then FCFS will produce the lowest average response time.

12. The Banker's algorithm can be used to help avoid deadlocks (caused by resource cycles) from occurring between a set of threads.

True False

- True. The Banker's algorithm double-checks each resource acquisition request to see if it might have the potential to cause deadlock and delays such acquisitions until they can be done safely.

13. In modern computing systems, page faults could be handled without OS involvement.

True False

- False. Only OS can handle a page fault.

14. Assuming high context-switching overhead, a good scheduling algorithm can achieve both high throughput and low latency for any mix of jobs.

True False

- False. High throughput is achieved by switching infrequently, whereas low latency is best achieved by interrupting long-running threads and switching to threads with the shortest burst.

15. A child process can communicate with its parent by utilizing a data structure on its heap that was allocated before the parent performed a fork() system call.

True False

- False. Since child and parent have different address spaces, their heaps are distinct.

Q2 (20 points) Multiple Select

1. (2.5 points) Select all true statements.

   ☒ A process's virtual address space can be larger than the size of the physical address space.
   ☒ The physical address space can be larger than a process's virtual address space.
   ☒ A process cannot access more memory than the size of its virtual address space.
   ☐ A process cannot access more memory than the size of the physical address space.
   ☐ You cannot use segmentation and paging at the same time.

2. (1.5 points) Select all ways that one can use to avoid/prevent deadlock

   ☒ Dynamically delay resource requests that would otherwise put the system in an unsafe state.
   ☒ Provision "infinite" units of each resource (i.e. more than the total demand by all threads).
   ☒ Only allow resources to be requested in a fixed order across all threads.

3. (2.5 points) Select all true statements under the assumption that pages are all the same size.

   ☒ A paging memory model may suffer from internal fragmentation.
   ☒ A segmentation memory model may suffer from external fragmentation.
   ☐ A single level page table dynamically increases in size when new pages are mapped.
   ☐ A paging memory model may suffer from external fragmentation.
   ☐ None of the above statements is true.

4. (2 points) Select all true statements.

   ☐ Read-modify-write operations cannot be performed in user mode for security reasons.
   ☒ The wait operation in a condition variable requires an implementation that atomically drops the lock and puts the calling thread on the corresponding wait queue.
   ☐ You can check the value of a semaphore and decide whether you want to wait or proceed.
   ☐ Calling `sem_wait` and `sem_wait` again immediately after results in undefined behavior.

5. (2.5 points) Select all true statements about POSIX threads (pthreads).

   ☒ When the last thread in a process exits, the process will terminate.
   ☐ Once `pthread_create` is called, the new thread will start running immediately.
   ☒ Once `pthread_join` is called on a sleeping thread, the calling thread will be blocked immediately.
   ☐ Once `pthread_join` is called, the thread being joined on will start running immediately.
   ☒ Once `pthread_yield` is called, the OS may reschedule the calling thread to run immediately.

6. (2.5 points) Which of the following are guaranteed to terminate the running user process? (choose all that apply)

   ☒ Receiving the `SIGKILL` signal
   ☐ Segmentation fault (i.e. receiving `SIGSEGV`)
   ☐ Interrupts
   ☐ Executing a trap instruction
   ☐ None of the above

7. (2.5 points) Select all that is true about sockets and IPC.

☒ pipe allocates two new file descriptors, one for reading and another for writing.
☒ A socket is a file descriptor which you can read from or write to.
☐ Processes on the same machine cannot communicate with each other through a socket.
☒ Threads in the same process can communicate with each other through a pipe.
☒ Typing CTRL + C in a UNIX shell is a form of interprocess communication.

8. (2 points) What are the disadvantages of disabling interrupts to serialize access to a critical section? (choose all that apply)

☒ User code cannot utilize this technique for serializing access to critical sections.
☒ This technique would lock out other hardware interrupts, potentially causing critical events to be missed.
☒ This technique is a very coarse-grained method of serializing, yielding only one such lock for each core.
☒ This technique could not be used to enforce a critical section on a multiprocessor.

9. (2 points) What are some disadvantages of Base&Limit style address translation? (choose all that apply):

☐ Base&Limit cannot protect kernel memory from being read by user programs.
☐ Context switches incur a much higher overhead compared to use of a page table.
☒ Base&Limit will lead to external fragmentation.
☐ With Base&, each process can have its own version of address “0”.

10. (2 points) Which of the following are true about condition variables? (choose all that apply):

☒ `cond_wait()`, and `cond_signal()` should only be used when holding the lock associated with the condition variable.
☐ `cond_signal()` can only be called after setting the boolean condition associated with the condition variable to true.
☒ For code readability, condition variables are generally preferred over semaphores.
☒ A thread that was previously blocked in `cond_wait()` and has received a signal may become blocked again before returning from `cond_wait()`.

11. (2 points) Which of the following are true? (choose all that apply):

☒ With a global memory replacement policy, one process could slow down all other processes.
☒ With a local memory replacement policy, one process cannot thrash other processes.
☒ Page fault rate could be used as a metric to detect thrashing.
☒ A local memory replacement policy might lead to inefficient memory allocation.

Q3. (22 points) Consider the following C program (assume all system calls succeed). In this code,

`int dup2(int oldfd, int newfd);`

is a system call that causes the file descriptor specified by `newfd` to refer to the same open file description as the descriptor `oldfd`. In other words, the file descriptor `newfd` is adjusted so that it refers to the same open file description as `oldfd`. If the file descriptor `newfd` was previously open, it is closed before being reused; the close is performed silently (i.e., any errors during the close are not reported by `dup2`). The steps of closing and reusing the file descriptor `newfd` are performed atomically. On success, the system call returns the new file descriptor. On error, -1 is returned. If `oldfd` is not a valid file descriptor, then the call fails, and `newfd` is not closed. If `oldfd` is a valid file descriptor, and `newfd` has the same value as `oldfd`, then `dup2` does nothing, and returns `newfd`.

Moreover, in this code, we use:

- `int fileno(FILE *stream)` system call to examines `stream` and returns the integer file descriptor used to implement it.
- `O_WRONLY` to open the file **write-only.**
- `O_CREAT` to **create the file if it does not exist.**
- `O_TRUNC` to if the file **already exists**, truncate it to **size 0** (erase its contents).
- `calloc` system call to allocate memory, like malloc except that it initializes all allocated bytes to 0.

```
01 void* rem(void *args) {
02   printf("Blue: %d\n", *((int*) args));
03   exit(0);
04 }
05 void* ram(void *args) {
06   printf("Pink: %d\n", ((int*) args)[0]);
07   return NULL;
08 }
09 int main(void) {
10   pid_t pid;
11   pthread_t pthread;
12   int status;
13   int fd = open("emilia.txt", O_CREAT|O_TRUNC|O_WRONLY, 0666);
14   int *subaru = (int*) calloc(1, sizeof(int));
15   printf("Original: %d\n", *subaru);
16   if(pid = fork()) {
17     *subaru = 1337;
18     pid = fork();
19   }
20   if(!pid) {
21     pthread_create(&pthread, NULL, ram, (void*) subaru);
22   } else {
23     for(int i = 0; i < 2; i++)
24       waitpid(-1, &status, 0);
25     pthread_create(&pthread, NULL, rem, (void*) subaru);
26   }
27   pthread_join(pthread, NULL);
28   if(*subaru == 1337)
29     dup2(fd, fileno(stdout));
30   printf("All done!\n");
31   return 0;
32 }
```

1. **(4 points)** Including the original process, what is the maximum and minimum number of processes and thread that run? Justify your answer.
   Max and min processes: 3
   Max and min threads: 6

2. **(6 points)** Provide all possible outputs in standard output. If there are multiple possibilities, put each in its own box. You may not need all the boxes.

Original: 0 Pink: 1337 Pink: 0 All done! Blue: 1337	Original: 0 Pink: 0 Pink: 1337 All done! Blue: 1337	Original: 0 Pink: 0 All done! Pink: 1337 Blue: 1337			

3. **(4 points)** Provide all possible contents of `emilia.txt`. If there are multiple possibilities, put each in its own box. You may not need all the boxes. (for blank file, write _blank_)

All done!					

4. **(4 points)** Suppose we deleted line 28. Provide all possible contests of `emilia.txt`. If there are multiple possibilities, put each in its own box. You may not need all the boxes. (for blank file, write _blank_)

All done! All done!					

Parent and child process both have the same file descriptor.

5. **(4 points)** What if, in addition to doing the change in part (4), we also move line 13 (where we open the file descriptor) between lines 19 and 20? (for blank file, write _blank_)

All done!	_blank_				

Parent and child have different file descriptors (they override each other's writes). Blank happens when the main process opens the file after the two children have terminated.

Q4. Deadlock (8 points) Suppose we have the following total resources:

Resource A	Resource B	Resource C
7	5	12

and threads T1, T2, T3, and T4 with the following current and maximum required allocations:

	Current Allocations			Maximum Required Allocations		
	A	B	C	A	B	C
T1	2	1	2	5	2	8
T2	0	2	1	3	2	3
T3	1	1	5	3	3	10
T4	1	0	2	2	2	4

1. (4 points) Is the system in a safe state? If so, provide a sequence of thread executions that would allow all threads to terminate. If not, explain why in two sentences or less.

Yes, the system is in a safe state. T2 -> T4 -> T3 -> T1

2. (4 points) Threads are allowed to acquire resources as they request them without any restriction, and as a result the system has entered the deadlocked state shown below.

	Current Allocations			Maximum Required Allocations		
	A	B	C	A	B	C
T1	3	1	2	5	2	8
T2	1	1	0	3	2	3
T3	2	2	7	3	3	10
T4	1	0	3	2	2	4

We can force a single thread to release all of its currently held resources. That thread then restarts from the beginning and may request resources again. Choose one thread to forcefully release all of its resources and restart so that all programs can eventually finish. If this is not possible (i.e., deadlock could still happen), write ``No thread." Write your answer in the box below. In either case, provide a short explanation outside the box (no more than two sentences).

Thread:

"No thread" because threads acquire resources as they request them. It is possible that the thread whose resources were forcefully released runs again without being context switched and reacquires the same set of resources, leading to the same deadlock state. This illustrates the importance of using algorithms such as Banker's algorithm to decide whether to grant resource requests.

Q5. Get a Pizza This! A new show hosted by a famous TV chef is being pitched, and you are asked to write a simulation of it. The show is about making pizza. A pizza requires three ingredients: dough, sauce, and cheese. All three ingredients are necessary to make a pizza (otherwise it does not meet the definition of a pizza). Each contestant has an unlimited supply of one ingredient. Contestant A has an unlimited supply of dough, Contestant B has an unlimited supply of sauce, and Contestant C has an unlimited supply of cheese. Each contestant needs to obtain the two ingredients they do not have in order to make a pizza. Contestants repeatedly attempt to make pizzas in a loop until time is up.

At the beginning of the episode, the host places two different random ingredients on the table. Contestants can signal the host to request more ingredients, but they should only do so if they need them. Each time the host is woken up (signaled), the host again places two different random ingredients on the table. When an ingredient is placed on the table, the host posts on the corresponding semaphore. For example, if the host places cheese and sauce on the table, the host posts on both `cheese` and `sauce`.

In this scenario, resources are provided by an external system (the host), and the contestants are processes that request resources. However, the system should not be wasteful: resources should only be requested when they are needed, and processes should take only what they need. A process should only wake up if it can perform useful work.

Consider the following solution. All semaphores are initialized to 0, except for `host`, which is initialized to 1 (so the host runs first).

Contestant A	Contestant B	Contestant C

`wait( sauce )`
`get_sauce()`
`wait( cheese )`
`get_cheese()`
`make_pizza( )`
`post( host )`
	`wait( dough )`
	`get_dough()`
	`wait( cheese )`
	`get_cheese()`
	`make_pizza( )`
	`post( host )`
		`wait( sauce )`
		`get_sauce()`
		`wait( dough )`
		`get_dough()`
		`make_pizza( )`
		`post( host )`

1. (4 points) Does this work? Explain.

No. Deadlock can easily occur. Suppose the host puts out sauce and dough. If contestant B takes the dough and contestant A takes the sauce, then both of them are blocked and nobody can proceed and nobody gets pizza.

2. (10 points) Now imagine that each contestant has a helper. The helper's job is to help their contestant make pizza by determining whose turn it is. For this purpose, there are Boolean variables `dough_present`, `sauce_present`, and `cheese_present`, all initialized to `false`. These variables are protected by a semaphore called `mutex`. The helpers update these variables and, based on the available information, signal which contestant should go to the table and take the ingredients. Each contestant now has a semaphore (e.g., `contestantA` for contestant A) that the helpers will post on. Contestants are still responsible for signaling the host to place more ingredients on the table. Complete the following lines to implement the helpers.

Helper 1	Helper 2	Helper 3

`wait( sauce )`
`wait( mutex )`
`if dough_present`
 	
`  dough_present = false;`
`  post( contestantC )`
 	
`else if cheese_present`
 	
`  cheese_present = false;`
`  post( contestantA )`
 	
`else`
 	
`  sauce_present = true;`
 	
`end if`
`post( mutex )`
	`wait( dough )`
	`wait( mutex )`
	`if sauce_present`
	
	`  sauce_present = false;`
	`  post( contestantC )`
	
	`else if cheese_present`
	
	`  cheese_present = false;`
	`  post( contestantB )`
	
	`else`
	
	`  dough_present = true;`
	
	`end if`
	`post( mutex )`
		`wait( cheese )`
		`wait( mutex )`
		`if dough_present`
		
		`  dough_present = false;`
		`  post( contestantB )`
		
		`else if sauce_present`
		
		`  sauce_present = false;`
		`  post( contestantA )`
		
		`else`
		
		`  cheese_present = true;`
		
		`end if`
		`post( mutex )`

3. (2 points) Given the helpers described in part (2), complete the code below for Contestant A.

```
wait( contestantA )
get_sauce()
get_cheese()
make_pizza( )
post( host )
```