Please print in pen:
Waterloo Student ID Number:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

WatIAM/Quest Login Userid:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

UNIVERSITY OF WATERLOO

# Examination
# Midterm
# Winter 2020
# CS 350

## Closed Book

Candidates may bring no aids (no calculators).

Times: Thursday 2020-03-05 at 19:00 to 20:50 (7 to 8:50PM)
Duration: 1 hour 50 minutes (110 minutes)
Exam ID: 4377126
Sections: CS 350 LEC 001-003
Instructors: Kevin Lanctot, Lesley Ann Istead

## University of Waterloo
## CS350 Midterm Examination

### Winter 2020

**Student Name:** _____

**Closed Book Exam**
**No Additional Materials Allowed**
**The marks for each question add up to a total of 109.**



Please initial:

Please initial:

**1. (14 total marks)** True or false.

| | |
|---|---|
| | (a) The trapframe is stored on the user stack. |
| | (b) A system call is an interrupt. |
| | (c) `ll` and `sc` are equivalent to `lw` and `sw`. |
| | (d) `rfe` causes a return from an interrupt. |
| | (e) `fork` returns 0 to the parent process. |
| | (f) A binary semaphore is identical to a lock in every way. |
| | (g) To force a thread to wait, use the function `thread_exit`. |
| | (h) Only `volatile` is needed to prevent race conditions in your code. |
| | (i) Parent and child processes share the same address space in OS/161. |
| | (j) A child process may call `waitpid` on its siblings without error. |
| | (k) The producers-consumers problem cannot be solved with just locks. |
| | (l) Compiler optimizations have no impact on the multi-threaded code. |
| | (m) The only reason to isolate the kernel from users is security. |
| | (n) Disabling interrupts is sufficient to prevent race conditions on any machine. |

**2. (6 total marks) Short Answer Part 1**

**a. (2 marks) Efficiency**

Which of the following would be more efficient and why?

i Calling `malloc` to create an array of 1000 ints during each loop iteration ($n$ iterations).

ii Creating a single array of 1000 ints on the heap once, and reusing this array for each iteration of the loop.

Answer:

Justification:

**b. (2 marks) Threads**

Give two consequences of a thread from a user process having only a single stack to be used by both the user program and the kernel.

Answer:

**c. (2 marks) Processes**

What are the benefits of using multiple threads in a single process to solve a problem over using multiple processes?

Answer:



Please initial:

**3. (10 total marks) Short Answer Part 2**

**a. (2 marks) Multiprogramming**
Multiprogramming improves CPU utilization. Explain how it achieves this.

Answer:

**b. (2 marks) Semaphores**
Give an example of a scenario where a binary semaphore would be more appropriate than a lock.

Answer:

**c. (6 marks) Condition Variables**
List, in order, the six steps of `cv_wait`.

Answer:



Please initial:

**4. (6 total marks) Threads**

Consider the following pseudocode:

Thread 1:

```
for i = 1 to N
    sleep for S units
    compute for 2C units
```

Thread 2:

```
for i = 1 to N
    compute for C units
    sleep for S units
```

Assume Thread 1 starts first. At what time, in terms of $C$, $S$, and $N$, do both threads finish if the quantum does not matter and $S = C$? Justify your answer.



Please initial:

**5. (8 marks)**

Consider the following functions that use locks and condition variables. Modify the code so that it uses only locks, but has the same functionality.

```
int volatile turn = 0;

void FuncA()
    lock_acquire( lock );

    while ( turn != 0 )
        cv_wait( cv0, lock );

    DoTaskA();

    turn = 1;
    cv_signal( cv1, lock );
    lock_release( lock );


void FuncB()
    lock_acquire( lock )

    while ( turn != 1 )
        cv_wait( cv1, lock );

    DoTaskB();

    turn = 0;
    cv_signal( cv0, lock );
    lock_release( lock );
```



Please initial:

**6. (12 marks)**

A user process was executing sort when it decided to call `getpid`. Draw the user and kernel stack for an OS/161 process that is preempted while executing `sys_getpid`. The interrupt handler for the clock is called `timer_interrupt_handler`.



Please initial:

**7. (15 marks)**

Consider a system that uses single-level paging for virtual memory with 32 bit physical and virtual addresses. Suppose page size 64KB ($2^{16}$ bytes).

**(a) (1 mark)** How many pages of virtual memory are there?

**(b) (1 mark)** How many frames of physical memory are there?

**(c) (1 mark)** How many bits are needed for the page offset?

**(d) (1 mark)** How many bits are needed for the page number?



Please initial:

**7 (continued).**

**(e) (1 mark)** A process uses a contiguous $2^{20}$ bytes of memory for its address space. How many valid entries will the page table have?

**(f) (10 marks)** What is the page number for each of the following virtual addresses? If the process described in (e) uses virtual addresses $[0, 2^{20})$, which of these addresses will be valid?

(i) 0xF00D 5555

(ii) 0xEA5E 0ACE

(iii) 0xC0DE C0DE

(iv) 0x0000 1234

(v) 0xEEEE EEEE



Please initial:

**8. (5 marks)**

Suppose we want to implement the system call `waitany(pid)` that causes the caller to wait on the specified process (by PID) to terminate. Unlike `waitpid`, `waitany(pid)` does not require the process to be the child—it can be any process.

**(a) (2 marks)** Describe how the implementation of `waitany(pid)` differs from `waitpid`.

**(b) (1 mark)** Given `waitany`, when can a process be fully deleted and its PID reused?

**(c)(2 marks)** If the process that calls `waitpid` or `waitany` has more than one thread, how could the sleep be handled?

**9. (8 marks)**

OS/161 does not allow user programs to fork new threads. What changes would be required to add this ability? You can assume the synchronizaton primitives lock, semaphores, and cv's have already been made and are available to user applications. You can also assume that process management calls, such as `fork`, have already been updated to handle multiple threads.

**10. (7 marks)**

Consider the following pseudocode implementation of a semaphore:

```
P( semaphore * s)
{
    KASSERT( s != null );
    spinlock_acquire( s->spinlock );
    while ( s->count < 0 )
    {
        spinlock_release( s->spinlock );
        wchan_lock( s->wchan );
        wchan_sleep( s->wchan );
        spinlock_acquire( s->spinlock );
    }
    s->count --;
    s->owner = curthread;
    spinlock_release( s->spinlock );
}

V( sempahore *s )
{
    KASSERT( s != null )
    spinlock_acquire( s->spinlock );

    count --;

    spinlock_acquire( s->spinlock );
}
```

Does this semaphore work? If yes, explain why. If no, correct it.



Please initial:

**11. (14 marks)**

Consider the following pseudocode:

```
const int width = 100;
const int height = 100;
bool * img;
lock * mutex;

void init() {
    lock_create( mutex );
    img = malloc( width * height * sizeof( bool ) );

    lock_acquire( mutex );
    for ( int i = 0; i < width; i ++ )
        for ( int j = 0; j < height; j ++ )
            img[i][j] = false;
    lock_release( mutex );
}

void do_something( void * itm, unsigned int num ) {
    lock_acquire( mutex );
            *(img + num) = magic_function( num );
    lock_release( mutex );
}

int main() {
    init();
    for ( int i = 0; i < width; i ++ )
        for ( int j = 0; j < height; j ++ )
            thread_fork( "", null, do_something, null, i + j * height );

    for ( int i = 0; i < width; i ++ ) {
        for ( int j = 0; j < height; j ++ ) {
            if ( img[i][j] ) printf( "X" );
            else printf( "O" );
        }
        printf( "\n" );
    }
}
```

Simplify, improve the performance of, and correct this code. Assume that `magic_function` does not access `img`. Assume that this code is a user program, and that `thread_fork`, etc., are implemented.



Please initial:

**12. (4 marks)**

Consider the following two functions `f()` and `g()` in a multi-threaded environment. The only function that `f()` calls is `g()` and the only function that `g()` calls is `thread_yield()`.

```
void f() {
  ⋮
  g();
  ⋮
}
```

```
void g() {
  ⋮
  thread_yield();
  ⋮
}
```

Both `f()` and `g()` make use of the register `s6`. All functions (including those in the kernel) follow the proper caller-save, callee-save format except that there is a bug in the creation of a switchframe so that the `s6` register never gets saved in the switchframe or restored from it. You may assume that `f()` and `g()` are part of a kernel program and have access to `thread_yield`.

**(a)** Describe a scenario where `f()`'s value stored in `s6` (for thread `T1`) gets modified outside the function `f()` or give an argument why it will never happen.

**(b)** Describe a scenario where `g()`'s value stored in `s6` (for thread `T1`) gets modified outside the function `g()` or give an argument why it will never happen.



Please initial: