University of Waterloo
CS350 Midterm Examination

Spring 2019

Student Name: ____________________

Closed Book Exam
No Additional Materials Allowed
The marks for each question add up to a total of 98

1. (14 total marks) True or false.

All **False**.
1 mark each.

	(a) Multiprogramming is the same thing as timesharing.
	(b) Threads share local variables.
	(c) Another thread must be used to preempt a thread that does not yield, block, or exit.
	(d) `switchframe_switch` and `trapframe` save every register.
	(e) `thread_yield` guarantees that a different thread will get the CPU.
	(f) The counter of a semaphore can have positive and negative values.
	(g) A condition variable has a wait channel and a spinlock.
	(h) A shared volatile variable does not need a lock to protect it.
	(i) It is more efficient to have one wait channel for the entire OS, than to have one wait channel for each item/resource.
	(j) `fork` creates a new process with an empty address space.
	(k) All processes share a single address space.
	(l) `execv` returns only if it succeeds.
	(m) Only variable addresses are virtual.
	(n) Dynamic relocation does not fragment physical memory.

2. (6 total marks)

a. (2 marks) Efficiency

Which is typically faster and why:

i Printing the numbers from 1 to 1000000, one number at a time.

ii Creating a string with the numbers from 1 to 1000000 and printing that string.

Answer:

Justification:

B is faster. A has 1 million system calls. B has one.

b. (2 marks) Spinlocks]

Spinlocks disable interrupts when they are acquired, then enable them when released. What are the consequences of this action?

Answer:

Answer: no preemption, no spinning (in single CPU environment)

c. (2 marks)

Which of the following operations could be executed without the OS ever being involved and why?

1. `printf`
2. `malloc`
3. adding two variables
4. `fopen`
5. none of the above

Answer:

Justification:

None of the above. printf and fopen require system calls, malloc and adding two variables may require the kernel

3. (7 total marks)

a. (2 marks) System calls

Which of the following would not be used/called to implement the `sleep(time)` system call and why?

1. the clock device
2. `thread_exit`
3. `switchframe_switch`

Answer:

Justification:

`thread_exit`, because the thread is blocking ... not exiting.

b. (3 marks) Race conditions

List three negative effects a race condition may have on a running program.

Answer:

incorrect result/output runtime error memory leak

c. (2 marks) virtual memory

Give one advantage and one disadvantage of letting the kernel translate every virtual address.

Advantage:

Disadvantage:

Adv: OS can implement vm any way it chooses Dis: performance

4. (4 total marks)

a. (2 marks)
Is it possible to be blocked on two or more wait channels at the same time? Why or why not?

Answer:

Justification:

No. If you block on wc 1, you are not running and cannot run, so you cannot then block on wc2.

b. (1 mark)
A computer has 4 CPUs. Each CPU has 16 cores. Each core can run 2 threads. How many threads can run in parallel?

4*16*2 = 128

c. (1 mark)
If a process calls `execv`, how does it affect that processes parent? It doesn't.

5. (7 marks)

Draw a **state diagram** (not a stack diagram) for threads. Be sure to indicate how a thread moves from one state to another.

6. (12 marks)

Draw the user and kernel stack for a process that is preempted while executing `sys__exit`. The interrupt handler for the clock is called `timer_interrupt_handler`.

ONE point each. user:

- some indication of other stack frames
- __exit

kernel:

- trapframe
- mipstrap
- syscall
- sys__exit
- trapframe
- mipstrap
- timer interrupt handler (or something similarly named, or mainbus and timer)
- thread yield
- thread switch
- switchframe

7. (20 marks)

Suppose a system uses segmentation as its implementation of virtual memory. Physical addresses are 64 bits and virtual addresses are 64 bits. Suppose there are 4 segments and the MMU uses limit and relocation registers instead of a segment table. Two bits are required for the segment number.

(a) (1 mark) What is the maximum size of any segment? 2^{62} bytes

(b) (15 marks) Given the following values for limit and relocation registers, fill in the table. Indicate **exception** in the physical address column if an exception would occur.

Segment 0 relocation: 0x7800 0000, limit: 0x1000

Segment 1 relocation: 0x1000 0000 0000 0000, limit: 0x4000 0000

Segment 2 relocation: 0x0, limit: 0x500

Segment 3 relocation: 0x1000, limit: 0x1500

Virtual Address	Segment	Offset	Physical Address
0xF00D F00D F00D F00D			
0xF000 0000 0000 000D			
0x1000			
0x1501			
0xA000 0000 0000 ACE0			

ONE point each entry.

0xF00D F00D F00D F00D - 3 - 0x300D FOOD FOOD FOOD - exception

0xF000 0000 0000 000D - 3 - 0x3000 0000 0000 000D - exception

0x1000 - 0 - 0x1000 - exception

0x1501 - 0 - 0x1501 - exception

0xA000 0000 0000 ACE0 - 2 - 0x2000 0000 0000 ACE0 - exception

7 (continued).

(c) (4 marks) A process has 12MB allocated for its stack segment and needs to double the size to 24MB. Assuming that we cannot expand the segment in place, what steps would be required to double the size?

ONE point each

- check and acquire 24mb space
- if does not exist, return error
- otherwise, copy old stack to new space
- update relocation and limit in both MMU and proc structure

8. (12 marks)

`kill(PID)` terminates the process with the specified PID, but only if the calling process has *administrator* rights, or, if the user that created the process is the same. Assume the kernel has a global process table with a function `lookup(PID)` that returns the process with the provided PID, or, returns NULL if no such process exists.

(a) (2 marks) What changes or additions to the kernels global structures will be required to support users and rights?

A table of users and their rights.

(b) (2 marks) What changes or additions to the process structure is required?

The user.

(c) (2 marks) What changes to `sys_fork` would be required?

Set the user of the new proc to be the user of the old proc.

(d) (6 marks) List the steps required to implement `sys_kill`.

- 1 mark get the process with PID or [1 mark] return error if none found
- get the user of the process with PID
- 1 mark if the user is the same as the user for this proc OR [1 mark] if the user of this proc has admin rights
- 1 mark set the exit status/code of proc PID to KILLED and signal waiting proc
- 1 mark delete address space and destroy proc with PID

9. (4 marks)

A readers-writers lock lets multiple threads into a critical section if all threads are just reading. But, if any one thread wants to write, then it must have mutual exclusion on the critical section.

Does the following pseudo code represent a working readers-writers lock? If yes, indicate why. If no, indicate why not.

```
struct rwlock {
    int readers;  // init 0
    int writers;  // init 0
    lock mutex;
    cv waiting;
};

read_acquire( rwlock * lk ) {
    lock_acquire( mutex );
        while ( writers != 0 ) cv_wait( mutex, waiting );
        readers ++;
    lock_release( mutex );
}

read_release( rwlock * lk ) {
    lock_acquire( mutex );
        readers --;
        if ( readers == 0 ) cv_signal( mutex, waiting );
    lock_release( mutex );
}

write_acquire( rwlock *lk ) {
    lock_acquire( mutex );
        while ( writers > 0 ) cv_wait( mutex, waiting );
        writers ++;
    lock_release( mutex );
}

write_release( rwlock *lk ) {
    lock_acquire( mutex );
        writers --;
        cv_broadcast( mutex, waiting );
    lock_release( mutex );
}
```

It does not work. Once there are no other writers, we don't check if any readers have snuck in. We have to then wait for all those readers to finish before we can enter the writer lock.

10. (6 marks)

Consider the following implementation of a condition variable. Assume that `cv_create` and `cv_destroy` have been implemented and are bug-free.

```
struct cv {
    wchan cv_wchan;
    lock * owner;
};

void cv_wait( lock *lk, cv * cv ) {
    kassert( lk != NULL );  kassert( cv != NULL );  kassert( lock_do_i_own( lk ) );
    wchan_lock( cv->cv_wchan );
    cv->owner = lk;
    lock_release( lk );
    wchan_sleep( cv->cv_wchan );
    lock_acquire( lk );
}

void cv_signal( lock *lk, cv * cv ) {
    kassert( lk != NULL );  kassert( cv != NULL );  kassert( lock_do_i_own( lk ) );
    kassert( owner == lk );
    wchan_wakeone( cv->cv_wchan );
}
```

This condition variable implementation is used in the following code. Assume appropriate create and destroy functions have been called.

```
volatile int count = 0;
cv notZero;
lock mutex, other;

void TakeOne( void * foo, unsigned long bar ) {
    lock_acquire( mutex );
        while( count == 0 )  cv_wait( mutex, notZero );
        count --;
    lock_release( mutex );
}

void MakeOne( void * foo, unsigned long bar ) {
    lock_acquire( mutex );
       count ++;
       if ( count > 0 ) cv_signal( mutex, notZero );
    lock_release( mutex );
}
```

10. (continued)

```
void MakeTwo( void * foo, unsigned long bar ) {
    lock_acquire( mutex );
        count += 2;
    lock_release( mutex );
    cv_signal( mutex, notZero );
}

void MakeThree( void * foo, unsigned long bar ) {
    lock_acquire( other );
        count += 3;
        cv_signal( other, notZero );
    lock_release( other );
}
```

What is the value of `count` after one thread executes `TakeOne` and one thread executes any of the following code. If any errors would occur, indicate the error and reason for its occurrence.

(a) (2 marks) `MakeOne`

0

(b) (2 marks) `MakeTwo`

Error. Assertion failure, lock is not owned in cv_signal.

(c) (2 marks) `MakeThree`

Error. Assertion failure, lock owned is NOT the lock used by the cv. ALSO ACCEPT - compile error, cv_signal is missing a parameter.

11. (6 marks)

Rewrite the following user code to minimize system calls. Pseudocode is acceptable. You may use a function `AppendToEnd( char * str, int num )`, which takes a large string buffer and appends a number to the back, separated by newlines. For example, if *str = "5*
n8
n", then `AppendToEnd(str, 100)` yields *str="5*
n8
n100". Note that `AppendToEnd` does not call malloc.

```
int main() {
    int rc = fork();
    if ( rc == 0 ) {
        int * counts = malloc(100 * sizeof( int ));
        CountValues( counts ); // sets array values to numbers between 0 and 100
        for ( int i = 0; i < 100; i ++ ) {
            printf( "%d\n", counts[i] );
        }
        _exit(0);
    } else {
        int ret;
        waitpid( rc, &ret, 0 );
    }
}
```

2 marks for removing the fork (because it did nothing of consequence)

2 marks for removing the malloc (because malloc can sometimes have a system call)

2 marks for creating a single string and printing ONCE

```
int main() {
    char str[400];  // no malloc, but may be a different size --- 11*100 = 1100 approx
    int counts[100]; // no MALLOC
    CountValues( counts );
    for (int i = 0; i < 100; i ++ ) {
        AppendToEnd(str, counts[i])
   }
   printf( "%s", str );
}
```