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.

1. (14 total marks) True or false.

They are ALL false. 1 mark each.

	(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 is faster.
A has potentially n system calls. B has one.

---

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:

---

the user program may gain access to kernel data left on the stack

to kernel may overflow the stack

ALSO ACCEPT any answer that is logical, including

could crash the kernel (unaligned memory access) if the stack pointer was not updated properly

they have different address spaces so access would be slower (e.g. copyin / copyout)

---

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:

---

shared memory access

access to complex synchronization

---

3. (10 total marks) Short Answer Part 2

a. (2 marks) Multiprogramming

Multiprogramming improves CPU utilization. Explain how it achieves this.

Answer:

when one process does an I/O operation
another job is selected and may run on the CPU

b. (2 marks) Semaphores

Give an example of a scenario where a binary semaphore would be more appropriate than a lock.

Answer:

the thread that acquires the resource
is not the thread that releases it
also...
when you want to start the system without letting any thread initially having access to a resource, e.g. initialization of a peripheral

c. (6 marks) Condition Variables

List, in order, the six steps of `cv_wait`.

Answer:

assert lock and cv are not null
assert that you own the lock
lock wchan
release lock
wchan sleep
acquire lock

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.

$3NC$ for correct answer T1: SCCSCCSCC ... SCC T2: CS-CS-CS- ... CS

for justification, which can be a drawing, whatever works

---

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 );
```

```
int volatile turn = 0;

void FuncA()
    lock_acquire( lock );

    while ( turn != 0 )
        lock_release( lock );
        /// may yield here
        lock_acquire( lock );

    ... do stuff ...

    turn = 1; [1 mark for removing signal]
    lock_release( lock );


void FuncB()
    lock_acquire( lock )

    while ( turn != 1 )
        lock_release( lock )
        // may yield here
        lock_acquire( lock )

    ... do stuff ...

    turn = 0; [1 mark for removing signal]
    lock_release( lock );
```

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`.

ONE point each. user:

- sort (or main)
- getpid

kernel:

- trapframe
- mips_trap
- syscall
- sys_getpid
- trapframe
- mipstrap
- timer interrupt handler (or something similarly named, or mainbus interrupt and timer/hardclock)
- thread_yield
- thread_switch
- switchframe

---

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?

$2^{32}/2^{16} = 2^{16}$

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

2^{16}

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

16

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

16

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?

$2^{20}/2^{16} = 2^4$

(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

---

i 0xF00D, exception

ii 0xEA5E, exception

iii 0xCODE, exception

iv 0x0000, valid

v 0xEEEE, exception

---

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`.

---

there are no changes to the locks/cv waiting (students do not need to mention this)

when calling `waitany`, no need to check if process is child

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

Only after waitany has been called on it.

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

---

there are two possible solutions, accept either

all threads sleep

only the thread that called wait sleeps

---

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.

---

add system call codes for fork, yield, exit

add thread fork, yield, and exit to the system call library

add calls to thread fork, yield and exit in the system call dispatcher (syscall)

assign appropriate, non-overlapping regions of the address space for new thread user staks

---

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.

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

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

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

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.

```
#define width 100
#define height 100
bool img[width][height];
lock * mutex;
volatile int count;

void init() {
    lock_create( mutex );

    [REMOVED LOCK]
    for ( int i = 0; i < width; i ++ )
        for ( int j = 0; j < height; j ++ )
            img[i][j] = false;

    count = 0;
}

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

    lock_acquire( mutex );
        count ++;
    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 );

    //waiting for all threads to finish
    ... a semaphore can be used instead, or a lock and cv combo

    lock_acquire( mutex );
       while ( count != width * height )
       {
           lock_release( mutex );
   thread_yield();
           lock_acquire( mutex );
       }
    lock_release( mutex );

   // replacing multiple prints with a single one
    char data[(width + 1) * height + 1]
    for ( int i = 0; i < width; i ++ ) {
        for ( int j = 0; j < height; j ++ ) {
            if ( img[i][j] ) data[i + j * height] = 'X';
            else data[i + j * height] = 'O';
        }
        data[i + j * height + 1] = '\n';
    }
    data[(width + 1) * height] = 0;
    printf( data );
}
```

if instead of making the char of bools local and filling it in inside main, they make it global and use the threads to do the work.

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.

`f()`'s value will be preserved despite the bug.

The only way that `f()` will not run before and after its call to `g()` will be because of preemption, in which case all of `f()`'s context will be stored in a trapframe.

Since `g()` is using proper callee-save protocol, it will save and restore `f()`'s value for `s6` so it will not be affected by the switchframe bug.

(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.

`T1` runs, calling `f()`, `g()` which stores a value in `s6`, then `thread_yield`. `T1` assumes `s6` will be preserved and another thread modifies `s6` (possibly saving the old value), then calls `thread_yield()`, also assuming its value in `s6` will be preserved. `T1` runs again $\Rightarrow$ `g()`'s value in `s6` has been modified.

This will fail if `thread_yield` and `thread_switch` don't use (and hence don't save and restore) `s6`.
