University of Waterloo
CS350 Midterm Examination

Fall 2019

Student Name: _____

Closed Book Exam
No Additional Materials Allowed
The marks for each question add up to a total of 95

1. (14 total marks) True or false.

a F

b F

c T

d F

e T

f F

g F

h F

i F

j T

k T

l T

m F

n F

1 mark each.

	(a) Disabling interrupts is sufficient to provide mutual exclusion in all situations.
	(b) `volatile` offers mutual exclusion on a variable.
	(c) An OS/161 lock is a mutex.
	(d) A semaphore's counter can be negative.
	(e) A process that is ready, blocked, or running must have at least one thread.
	(f) When a thread is forked (created), it is given a new address space by the kernel.
	(g) `yield` guarantees a context switch to a different thread.
	(h) The kernel uses the same stack as the user program.
	(i) In OS/161 user applications can directly call `sys_fork`.
	(j) System calls can be interrupted.
	(k) Unprivileged instructions can be executed by the CPU while it is in privileged mode.
	(l) Dynamic relocation suffers from fragmentation.
	(m) Processes can only communicate to each other using sockets.
	(n) Page tables only contain valid entries.

2. (6 total marks) Short Answer Part 1

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) Threads

Why do threads have a user and a kernel stack?

Answer:

1 mark to separate unprivileged and privileged information (security) **1 mark** to prevent the kernel from overflowing the users stack **ALSO ACCEPT** any answer that is logical

c. (2 marks) Processes

What are the benefits of using multiple processes to perform a task instead of multiple threads?

Answer:

2 marks isolation, OR **1 mark** errors do not impact each other AND **1 mark** processes can run different programs

3. (6 total marks) Short Answer Part 2

a. 2 marks) System calls

In OS/161, `waitpid` can only be called on your child process. When is it safe to reuse a PID in OS/161?

Answer:

1	**mark** when a terminating process has no living parent
1	**mark** after waitpid has been called on a process

b. (2 marks) Semaphores

What is a barrier semaphore used for? Give an example (text) of a situation where it might be used.

Answer:

1	**mark** forcing a thread to wait until all others have completed
1	**mark** accept any reasonable answer, typically will be "waiting for threads to finish to collect/aggregate results"

c. (2 marks) System Calls

The system call `kill` is used to terminate a process by PID. Should an arbitrary user or process be able to kill another process? If yes, explain why. If no, suggest which users and processes should be able to kill another process.

Answer:

1	**mark** NO
1	**mark** caller has kernel privilege, or, caller is owner of process

4. (6 total marks) Threads

Draw and label a thread state diagram that shows how threads move from one state to another.

1	**mark EACH** for STATES: running, ready, blocked
1	**mark EACH** for labeled transitions between those states—see thread notes

5. (5 marks)

Wait morphing is a technique used to improve the performance of certain synchronization primitives. It works by moving a thread from one wait queue directly to another without having to wake up, only to go to sleep immediately on another resource. Give the pseudocode for `cv_signal` that uses wait morphing. You may assume that wait channels have a public unlock, push, and pop function in addition to those provided by OS/161.

1	**mark** assert lock non null
1	**mark** checks lock ownership
1	**mark** does NOT call wakeone
1	**mark** pops thread from wchan
1	**mark** pushs thread onto lock wchan

6. (12 marks)

Draw the user and kernel stack for an OS/161 process that is preempted while executing `sys__fork`. The interrupt handler for the clock is called `timer_interrupt_handler`.

ONE point each. user:

- some indication of other stack frames
- __fork

kernel:

- trapframe
- mipstrap
- syscall
- sys__fork
- trapframe
- mipstrap
- timer interrupt handler (or something similarly named, or mainbus and timer)
- 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 16KB (2^{14} bytes).

(a) (1 mark) How many pages of virtual memory are there? $2^{32}/2^{14} = 2^{18}$

(b) (1 mark) How many frames of physical memory are there? 2^{18}

(c) (1 mark) How many bits are needed for the page offset? 14

(d) (1 mark) How many bits are needed for the page number? 18

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? $2^{20}/2^{14} = 2^6$

(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) 0x5555 5555

(ii) 0xEA5E 0ACE

(iii) 0x0000 1000

(iv) 0x0000 ABCD

(v) 0x0005 EEEE

i 01 0101 0101 0101 0101 - 0x15555 - exception/not-valid

ii 11 1010 1001 0111 1000 - 0x3A978 - exception/not-valid

iii 0000 0000 0000 0000 00 - 0x0 - valid

iv 00 0000 0000 0000 0010 - 0x2 - valid

v 00 0000 0000 0001 0111 - 0x17 - valid

8. (4 marks)

Consider the following implementation of `try_acquire`, which returns false instead of blocking when the lock is not available, and true otherwise. Does the implementation work? If yes, explain why. If no, fix it.

```
bool try_acquire( lock *lk ) {
    if ( lk->owner != NULL ) return false;
    else lock_acquire( lk->owner );
    return true;
}
```

1	**mark** It does not work.
1	**mark** asserting that lock is not null
1	**mark** acquring/releasing locks spinlock
1	**mark** replacing lock_acquire with setting owner

9. (10 marks)

Program A is executed. Draw the process tree. In each node, indicate what program is running and what value will be passed to `_exit`.

```
// Program A:
int x = 5;
int main() {
    int pid = fork();
    if ( pid == 0 ) {
        x = 2;
        execv( "B", "2" ); }
    else {
        waitpid( pid );
        _exit( x );
    }
}

// Program B:
int x = 0;
int main( int argv, char ** argc ) {
    int pid = fork();
    if ( pid == 0 ) execv( "B", "1" );
    else _exit( x );
    _exit( 1 );
}
```

10. (11 marks)

Suppose that when a process forks, instead of creating a completely new address space, the child process will initially share the code and data sections of the parents addres space to save time/memory. Suppose also that the processes use paged virtual memory. However, both parent and child processes should still be isolated.

(a) (2 marks) At what point would the two processes not be able to share pages of the code segment in the address space?

2	**marks** if a process calls execv

(b) (3 marks) At what point would the two proceses not be able to share a given page of the data segment in the address space?

2	**marks** if a process calls execv
1	**mark** if a process writes to a page of the data segment

(c) (2 marks) How would the OS detect a write to a page that is shared?

1	**mark** mark shared pages as read only
1	**mark** mmu throws exception on attempt to write to read-only page

(d) (4 marks) List the steps taken by the OS to resolve a process writing to a shared page.

1	**mark** allocate a new frame of physical memory
1	**mark** copy the contents to the new frame
1	**mark** update the proceses page table to indicate that the page maps to a different frame
1	**mark** mark both page table entries (each process) as read-write

11. (6 marks)

Consider the following pseudocode:

```
int array_sum( int * array, int n ) {
      if ( n == 0 ) return array[n];
      return array[n - 1] + array_sum( array, n - 1 );
}

int main(int argv, char ** argv) {
    int pid = fork();
    if ( pid == 0 ) {
        // 0 < argv[0] < 1000
        int * array = malloc( atoi( argv[0] ) * sizeof( int ) );
        for ( int i = 0; i < atoi( argv[0] ); i ++ ) array[i] = i % 3;
        int sum = array_sum( array, atoi( argv[0] ) );
        for ( int i = 0; i < sum; i ++ ) printf( \%d\n", i );
        _exit(0);
    }
    else {
        if ( waitpid( pid, ... ) )
            _exit(0);
    }
    return 0;
}
```

Improve the performance of this code. The output of the program must remain the same. You may use a function `append_num(char *txt, int num)`, which appends `num` to the end of `txt` such that each number is separated by a newline.

1	**mark** remove fork/waitpid/exit
1	**mark** replace malloc with static array
1	**mark** replace recursion with iteration
1	**mark** create fixed-size char array (no malloc)
1	**mark** call append_num in loop instead of printf
1	**mark** single printf call with large string