University of Waterloo
CS350 Midterm Examination

Fall 2018

Student Name: _____

Closed Book Exam
No Additional Materials Allowed

1. (15 total marks)

a. (2 marks)

Is it possible to have more than one switchframe in a kernel stack? Explain why or why not.

No. A running thread stores a switchframe in its kernel stack as it context switches to a different thread. Upon returning to the thread, the switchframe is popped off the thread's kernel stack. Since it is not possible for a thread to context switch away twice without context switching back once in between, there can only be one switchframe in a kernel stack.

b. (3 marks)

Explain why the following implementation of semaphore P is incorrect. Provide an example interaction between two threads that illustrates the problem.

```
void P(struct semaphore* sem) {
    spinlock_acquire(&sem->sem_lock);
    while (sem->sem_count == 0) {
        spinlock_release(&sem->sem_lock);
        wchan_lock(sem->sem_wchan);
        wchan_sleep(sem->sem_wchan);
        spinlock_acquire(&sem->sem_lock);
    }
    sem->sem_count--;
    spinlock_release(&sem->sem_lock);
}
```

wchan_lock should be called before spinlock_release. In this incorrect implementation, there is a small window in which no locks are held. During this window, the semaphore state can be changed by another thread.

Example interaction: Thread T1 is preempted in between spinlock_release and wchan_lock. Thread T2 executes semaphore V to completion. Upon returning to T1, the thread will sleep on a wait channel even though the semaphore count is larger than 0.

c. (2 marks)

Explain why system calls need to increment the EPC by 4 before returning to user space. Why is incrementing the EPC not necessary when handling other exceptions?

syscall is an instruction, and so returning to the PC of the system call will cause another syscall exception to be raised. With other exceptions, the exceptional circumstance should be resolved, and the command itself rerun.

d. (2 marks)

Explain why a page table entry does not contain a page number, yet a TLB entry contains both a page number and a frame number.

The page number is the index of the page table array, so it would be redundant to also store it. The TLB is a partial cache, and so must indicate which entries are cached.

e. (2 marks)

A trapframe contains more information than a switchframe. Why?

switchframe_switch is a function, and so has the usual calling conventions of a function; in particular, caller-saved variables may be trashed. An exception is not called, but simply raised unexpectedly, and so there are no calling conventions, so everything must be saved.

f. (4 marks)

Imagine a version of OS/161 with the following bug: When the exception caused by division by zero is raised, the kernel instead handles it as a system call. What will be the behavior of the following program if the compiler stores `n` in v0 and `d` in a0? What will be the behavior if the compiler stores `n` in a0 and `d` in v0?

Table of system calls:

System Call	**System Call #**
pid_t fork(void)	0
pid_t vfork(void)	1
int execv(const char *program, char **args)	2
void _exit(int exitcode)	3
pid_t waitpid(pid_t, int *status, int options)	4
pid_t getpid(void)	5

```
int main() {
    int n = 6;
    int d;
    for (d = 2; d >= 0; d--)
        n /= d;
    printf("%d\n", d);
    return 0;
}
```

First case: The program will exit with exit code 0.
Second case: The program will spawn children in a (slow) loop, with each child printing -1.

2. (8 total marks)

Consider the following functions:

```
int a, b;
struct lock* lock_a;
struct lock* lock_b;
struct cv* cv;

void funcAB() {
    lock_acquire(lock_b);
    // Code that reads/writes b
    lock_acquire(lock_a);
    // Code that reads/writes a
    while (b > 0) {
        cv_wait(cv, lock_b);
    }
    // More code that reads/writes b
    lock_release(lock_a);
    lock_release(lock_b);
}

void funcA() {
    lock_acquire(lock_a);
    // Code that reads/writes a
    lock_release(lock_a);
}

void funcB() {
    lock_acquire(lock_b);
    b = 0;
    cv_signal(cv, lock_b);
    lock_release(lock_b);
}
```

In this program, multiple threads concurrently call each of these functions. No other functions acquire or release `lock_a` and `lock_b`, and a thread will only access the global variables `a` and `b` if they are holding `lock_a` and `lock_b` respectively. You can assume the locks and condition variable have been created successfully. The global variables in this program do not have to be declared as volatile.

a. (2 marks)

What **concurrency problem** does this program suffer from?

Deadlock

b. (4 marks)

Provide a sequence of events that can trigger this problem.

Thread 1 calls funcAB and goes to sleep in cv_wait. Thread 2 calls funcB. Thread 3 calls funcAB and sleeps waiting for lock A, after acquiring lock B. Thread 1 awakes and attempts to take lock B.

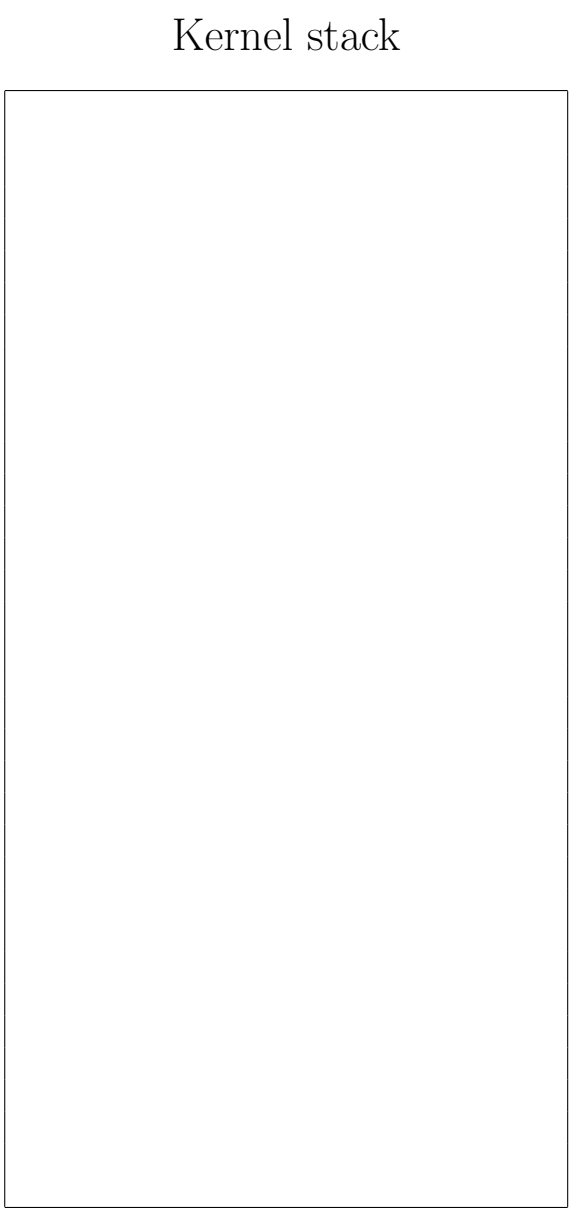
c. (2 marks)

What changes to any of the above functions would address this problem?

Acquire lock A before lock B in funcAB.

3. (6 marks)

An OS/161 process P accesses a virtual memory address which has a TLB miss. During `vm_fault` in the kernel, a timer interrupt fires and P gets preempted. Draw the relevant stack frames for P's kernel stack.



```
---
trapframe
---
mips_trap
---
vm_fault
---
trapframe
---
mips_trap
---
mainbus_interrupt
---
thread_yield
---
thread_switch
---
switchframe
---
```

4. (10 total marks)

You are designing a paged virtual memory system on a system with 32-bit virtual addresses and 48-bit physical addresses. Each page is 4KB (2^{12} bytes), and each page table entry is 64 bits (8 bytes, 2^3 bytes).

a. (2 marks)

In a single-level paged system, how many bits of a virtual memory address would refer to the page number and how many to the offset? How many bits of a physical address would refer to the frame number and how many to the offset?

20, 12, 36, 12

b. (2 marks)

How many bytes would a single-level page table require?

2^{23}

c. (2 marks)

If a page table entry contains a frame number, a valid bit, a writeable bit, and a single bit for tracking page useage, how many bits per page table entry are unused?

$64 - 36 - 3 = 25$

d. (2 marks)

How many page table entries fit onto a single page?

$2^9 = 512$

e. (2 marks)

What is the minimum number of levels necessary to implement a multi-level paged system if each page table at each level must fit into a single page?

3

5. (8 marks)

Consider the following code:

```
int waiter(int pid) {
    int rv, ec;
    if (pid != 0) {
        rv = waitpid(pid, &ec, 0);
        if (rv != -1 && WIFEXITED(ec)) {
            return WEXITSTATUS(ec);
        }
        return 0;
    }
    return 2;
}

int main() {
    int a, b;
    int res = 0;
    a = fork();
    b = fork();
    res += waiter(a);
    res += waiter(b);
    printf("%d\n", res);
    _exit(1);
}
```

Draw the tree of processes created, with the value printed for `res` in each node of the tree, and the vertices pointing from parents to children.

```
2->3->4
 \
  2
```

6. (8 total marks)

Many operating systems implement a system call `vfork`, which is similar to `fork`, but with these two changes:

1. The child process shares the parent process's address space. That is, the address space is *not* copied, it is shared.
2. Upon calling vfork, the parent process blocks until the child process has called `execv`.

a. (3 marks)

Change the following sketch of an implementation of `fork` to instead implement `vfork`. Assume that the appropriate changes are made to `execv` elsewhere. You may cross out steps and add new steps.

```
sys_fork() {

    create process structure

    copy address space

    choose pid for new process

    create parent/child relationship

    duplicate trapframe

    create thread for new process (running child_process)

}

child_process() {

    copy trapframe to stack

    modify trapframe

    enter usermode

}
```

Replace "copy address space" with "link to same address space", and add "block until child has reached execv".

b. (3 marks)

Each change in the semantics of `vfork` impacts the implementation of `execv`. How would an implementation of `execv` need to differ to support both of these changes in `vfork`?

If process was created with vfork, do not destroy the address space. Signal the parent.

c. (2 marks)

The documentation for `vfork` states that the child process shouldn't return from the function that called `vfork`. Why not?

The userspace stack is shared, so if the stackframe is popped, the behavior of the function when the parent returns is unpredictable.

7. (16 total marks)

You have been hired by the city of Waterloo to help solve a modified version of the "traffic intersection" problem for an intersection with significant pedestrian traffic. For safety, instead of allowing vehicles to share the intersection with pedestrians, you are to build a `scramble intersection` that, under specific conditions, stops vehicular traffic from all directions to allow pedestrians to cross the intersection.

In this problem, pedestrians and vehicles must never be inside the intersection at the same time. Vehicles must also not collide with other vehicles. When a pedestrian arrives at the intersection, he/she must wait until the intersection is clear of vehicles before entering. However, no additional vehicles must be allowed to enter the intersection while a pedestrian is either waiting or inside the intersection.

The intersection consists of two one-way roads: one north-to-south and the other east-to-west. Each vehicle arrives at the intersection from one of two directions (north or east), called its origin. It is trying to pass through the intersection and exit in the opposite direction of its origin, called its destination. **Turns for vehicles are not allowed**. Because pedestrians cannot collide with other pedestrians and can only cross when there are no vehicles inside the intersection, we do not need to know a pedestrian's origin or destination.

Implement the following six functions. Global variables can be defined in the provided space. Your solution should be efficient. It should also prioritize pedestrians while providing fairness between vehicles.

```
// Define your global variables here.
enum Directions {
    north = 0, east = 1
};
typedef enum Directions Direction;

Direction opposing_dir(Direction d) {
    return (d == north) ? east : north;
}

// This is a sample solution that follows a fairly simple solution approach.
// There are many correct solutions to this problem.  Note that we did not
// require a "perfectly" fair solution. For example, there was no grade
// deduction if your solution woke up vehicles from a random non-empty
// direction after the last pedestrian exits the intersection. This solution
// has never been compiled; there may be syntax errors.

struct lock* lk;
struct cv* dir_cv[2];
struct cv* ped_cv;
int cars_inside[2];
int cars_waiting[2];
int ped_inside;
int ped_waiting;
Direction next_after_ped;
```

```
// Called only once before starting the simulation.
void intersection_sync_init(void) {
    lk = lock_create("lock");
    for (int i = 0; i < 2; ++i) {
        cars_inside[i] = 0;
        cars_waiting[i] = 0;
        dir_cv[i] = cv_create("cv");
    }
    ped_inside = 0;
    ped_waiting = 0;
    ped_cv = cv_create("cv");
    next_after_ped = north; // arbitrary
}

// Called only once at the end of the simulation.
void intersection_sync_cleanup(void) {
    for (int i = 0; i < 2; ++i) {
        cv_destroy(cars_inside[i]);
    }
    cv_destroy(ped_cv);
    lock_destroy(lk);
}
```

```
// Note that this solution is not perfectly fair. There are cases
// where a barging pedestrian can cause an unfair direction to be
// selected. A "perfectly" fair solution can be implemented
// using a ticket-based strategy.
void intersection_before_vehicle_entry(Direction origin) {
    lock_acquire(lk);
    cars_waiting[origin]++;
    // If there are cars waiting in the opposite direction,
    // wait for those cars to enter first. Note this is
    // only to introduce an order, and not for safety.
    if (cars_waiting[opposing_dir(origin)] > 0) {
        cv_wait(dir_cv[origin], lk);
    } else {
        // If the are no cars in the opposing direction,
        // then set next_after_ped to this direction. This
        // means that if a pedestrian causes the car to
        // wait, the pedestrians will know which cars to
        // wakeup when the last pedestrian exits.
        next_after_ped = origin;
    }
    // This car must wait if there are pedestrians inside
    // the intersection or waiting to enter. It must also
    // wait if there are cars from the opposing direction
    // inside the intersection.
    while (ped_inside > 0  ||
           ped_waiting > 0 ||
           cars_inside[opposing_dir(origin)] > 0) {
        cv_wait(dir_cv[origin], lk);
    }

    cars_waiting[origin]--;
    cars_inside[origin]++;
    lock_release(lk);
}

// Called by a vehicle after the vehicle leaves the intersection.
void intersection_after_vehicle_exit(Direction origin)
    lock_acquire(lk);
    if (--cars_inside[origin] == 0) {
        if (ped_waiting > 0) {
            cv_broadcast(ped_cv, lk);
        } else {
            // Wakeup opposing direction. Set current direction
            // as the next to wakeup after pedestrians.
            cv_broadcast(dir_cv[opposing_dir(origin)], lk);
            next_after_ped = origin;
        }
    }
    lock_release(lk);
}
```

```
// Called by a pedestrian before the pedestrian enters the intersection.
void intersection_before_pedestrian_entry(void) {
    lock_acquire(lk);
    ped_waiting++;
    while (cars_inside[north] > 0 ||
           cars_inside[east] > 0) {
        cv_wait(ped_cv, lk);
    }
    ped_waiting--;
    ped_inside++;
    lock_release(lk);
}

// Called by a pedestrian after the pedestrian leaves the intersection.
void intersection_after_pedestrian_exit(void) {
    lock_acquire(lk);
    if (--ped_inside == 0) {
        if (cars_waiting[next_after_ped] > 0) {
            // Wakeup and switch the next direction to wakeup
            // after pedestrians.
            cv_broadcast(dir_cv[next_after_ped], lk)
            next_after_ped = opposing_dir(next_after_ped);
        } else {
            // The next direction has no cars waiting. Wakeup
            // the opposing direction. There is no need to change
            // the next direction to wakeup after pedestrians.
            cv_broadcast(dir_cv[opposing_dir(next_after_ped)], lk);
        }
    }
    lock_release(lk);
}
```