University of Waterloo
CS350 Midterm Examination

Fall 2017

Student Name: _____

Closed Book Exam
No Additional Materials Allowed

1. (13 total marks)

a. (2 marks)

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

An exception may be interrupted, and both will push a trapframe.

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);
}
```

The transfer from the semaphore's spinlock to the wait channel's lock is not atomic. If thread 1 is preempted between spinlock_release and wchan_lock, thread 2 may call V(), waking no threads, before thread 1 goes to sleep, leaving thread 1 waiting although the value has already been changed.

c. (2 marks)

When an exception occurs, the CPU saves the contents of the `PC` to the `EPC`, turns off interrupts, and performs two other actions. List those two other actions.

- Switches to kernel privilege.
- Jumps to the exception handler.

d. (2 marks)

For what reasons would `waitpid` return an error?

If the pid does not refer to a real process, or the process it refers to is not the child of the current process. (Or, the process has already exited and been waited for; technically, this is a sub-case of the other two.)

e. (2 marks)

A trapframe contains more information than a switchframe. Why?

Switchframes are used during `thread_yield`, an intentional call to a function. The convention for function calls is allowed to lose some registers. Trapframes are used during interrupts, which are not intentional calls, so nothing may be lost.

f. (2 marks)

Why does `thread_fork` require a function for the thread to call, while `fork` is able to simply return in both processes?

The stack is stored in virtual memory, which is duplicated when creating a new process, allowing the continuing state to evolve differently. With a thread, virtual memory is not duplicated, and so the same stack cannot be used. Thus, a new stack must be created, and a function must be called to start off that stack.

2. (8 total marks)

Suppose we want to count the number of times the word "the" appears in Wikipedia. After downloading all the articles, the following program is written to perform this task:

```
char **articles;
int counts[NUMARTICLES];
struct lock *mutex;

void Count(char *word, int articleNum)
{
    int wordCount;
    [ ... find word count ... ]
    counts[articleNum] = wordCount;
}

int CountWord(char **articles, int articleCount)
{
    int i;
    for (i = 0; i < articleCount; i++)
        thread_fork([...], Count, word, i);

    int sum = sum(counts);
    printf("Number of words: %d\n", sum );
}
```

However, when this program is run, the number printed is incorrect.

a. (2 marks)

Why is the sum incorrect?

The forked threads may not have finished when the sum of the counts is calculated.

b. (6 marks)

Fix the code.

```
char **articles;
int counts[NUMARTICLES];
struct lock *mutex;

void Count(char *word, int articleNum)
{
    int wordCount;

    [ ... find word count ... ]

}

int CountWord(char **articles, int articleCount)
{
    int i;

    for (i = 0; i < articleCount; i++)
        thread_fork([...], Count, word, i);

    int sum = sum(counts);
    printf("Number of words: %d\n", sum );
}
```

```
char **articles;
int counts[NUMARTICLES];
struct lock *mutex;

struct semaphore *sem;

void Count(char *word, int articleNum)
{
    int wordCount;
    [ ... find word count ... ]
    V(sem);
}

int CountWord(char **articles, int articleCount)
{
    int i;

    sem = sem_create([...], 0);

    for (i = 0; i < articleCount; i++)
        thread_fork([...], Count, word, i);

    for (i = 0; i < articleCount; i++)
        P(sem);

    int sum = sum(counts);
    printf("Number of words: %d\n", sum );
}
```

3. (8 total marks)

A system uses 64-bit physical addresses and 48-bit virtual addresses. The page size is 2^{32} bytes. The system uses a single-level page table as its implementation of virtual memory.

a. (1 mark) How many page table entries does the page table contain?

$\frac{2^{48}}{2^{32}} = 2^{16} = 65,536$

b. (1 mark) How many bits are used for the page number?

16

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

32

d. (1 mark) If each page table entry is 128 bits ($2^4 = 16$ bytes), how big is a page table?

$2^{16} * 2^4 = 2^{20} = 1,048,576 = 1$ mebibyte ≈ 1 megabyte

e. (2 marks) Process P has a 256KB (2^{28} byte) contiguous address space, starting at the beginning of a page. How many pages are valid?

1. This system is really weird.

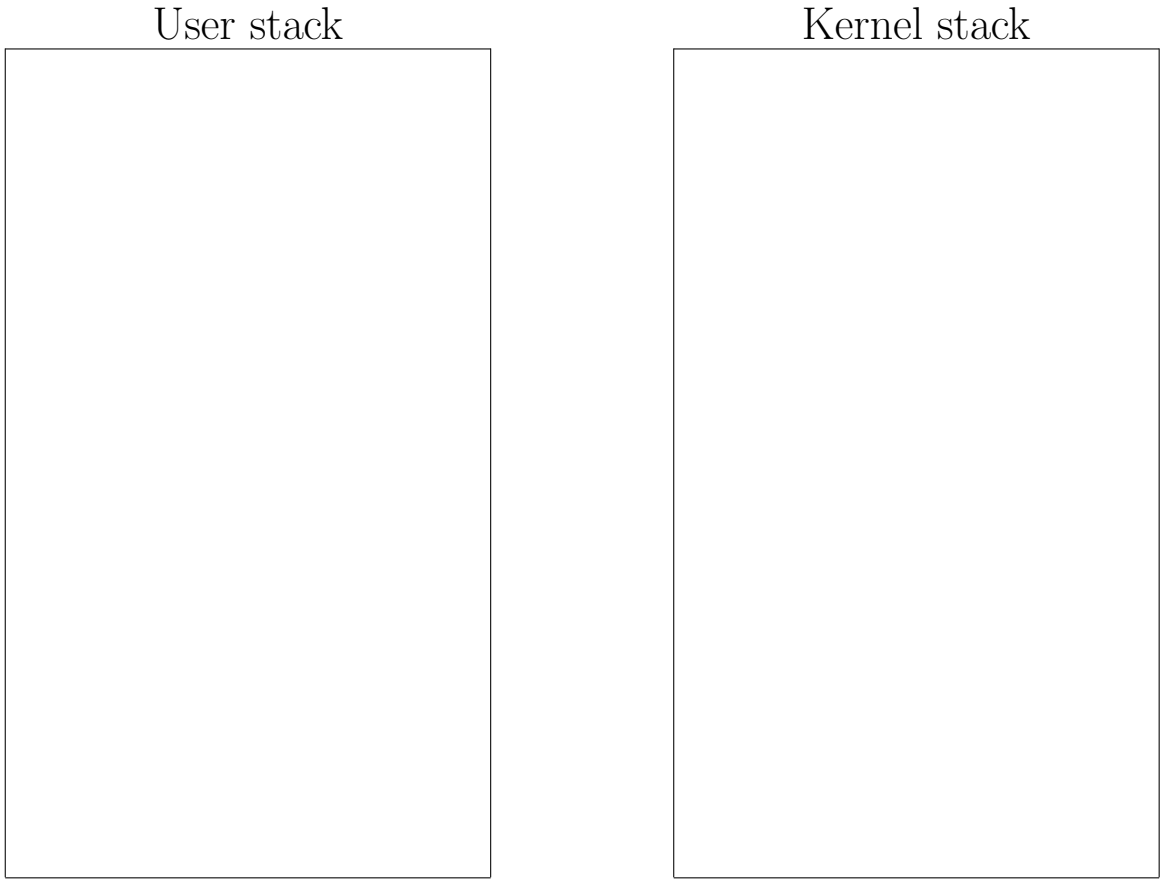
f. (2 marks) How much memory on the pages assigned to process P is unused?

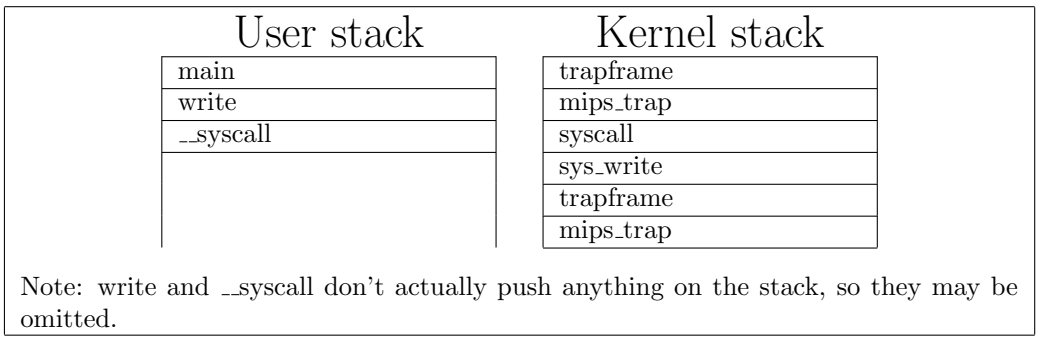
$2^{32} - 2^{28}$ bytes, almost the entire 4GB

4. (6 marks)

An OS/161 process P is in `sys_write`, from a `write` system call, when a timer interrupt fires. Draw the relevant stack frames for P's user and kernel stack, during the call to `mips_trap` caused by the timer interrupt. The application source code is as follows:

```
int main() {
    write(1, "Hello, world!\n", 8);
}
```





Note: write and __syscall don't actually push anything on the stack, so they may be omitted.

5. (8 total marks)

a. (2 marks)

Consider a single-level paging-based virtual memory system. Explain the consequence of having two valid page table entries from two different page tables store the same frame number.

The page is shared, so writes to the relevant virtual memory page in the first process are visible to reads from the relevant virtual memory page in the second process and vice-versa. In theory the processes can communicate information through this page.

b. (4 marks)

At this point in the course, a parent and child process can only communicate with each other through the exit code of the child process. For this question, explain in detail what changes you would need to make to the `fork` system call such that, in addition to its normal function, fork would allocate a shared region of memory that is readable but not writeable by the parent, and readable and writeable by the child. You can assume the kernel is using a paging-based virtual memory system. The interface to this modified fork system call is as follows:

```
pid_t fork(size_t size, void **retval);
```

Example code that allocates 4 KB of shared memory using this modified system call:

```
void *shared_memory;
rc = fork(4096, &shared_memory);
```

We are only interested in the changes that you need to make inside the kernel.

After copying the address space, it would be necessary to allocate the requested number of frames and assign them to unused pages in both processes. It would be convenient but not necessary to map them to the same unused pages.

c. (2 marks)

When would it be safe for the child process to write data to the shared memory region, and for the parent process to read data from the shared memory region? Explain your reasoning.

waitpid can be used for synchronization, so if the parent never reads until waitpid returns, then anything the child has written will always be finished before the parent reads it.

6. (8 total marks)

Draw the tree of processes for the following programs, in which `a.c` is compiled into `./a` and `b.c` is compiled into `./b`, and the root process starts by running `./a`. Include arrows from parents to children. For each process, indicate which program it's running at exit and its exit code.

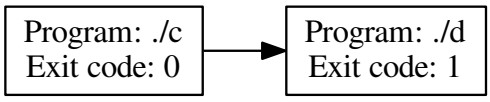
a.c:

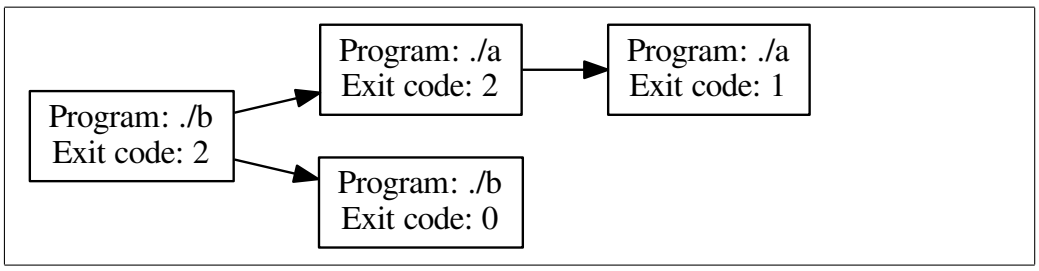
```
int main() {
    char buf[32];
    int p = fork();
    if (p != 0) {
        char *args[3];
        sprintf(buf, "%d", p);
        args[0] = "./b";
        args[1] = buf;
        args[2] = NULL;
        execv("./b", args);
    } else {
        if (fork() == 0)
            _exit(1);
        else
            _exit(2);
    }
    _exit(3);
}
```

b.c:

```
int main(int argc, char **argv) {
    int r = 0;
    int p = fork();
    if (p != 0) {
        int e;
        waitpid(p, &e, 0);
        if (WIFEXITED(e))
            r = WEXITSTATUS(e);
        waitpid(atoi(argv[1]), &e, 0);
        if (WIFEXITED(e))
            r += WEXITSTATUS(e);
    }
    _exit(r);
}
```

Example:





7. (16 total marks)

As part of Assignment 1, you implemented locks and condition variables using spinlocks and wait channels. In this problem, you must use locks and condition variables to build a Readers-Writer lock, a synchronization primitive that allows multiple concurrent readers to simultaneously read from shared data, or one writer to write to shared data.

A reader acquires and releases a Readers-Writer lock by passing READ as the first argument to `rwlock_acquire` and `rwlock_release`. Similarly, a writer passes WRITE as the first argument to `rwlock_acquire` and `rwlock_release`. Multiple readers must be able to simultaneously acquire the same lock if a writer does not already own (having acquired but not yet released) the lock, and no writers are waiting to acquire the lock. A writer must be able to acquire a lock if no readers or other writers currently own the lock. Only one writer can acquire the lock at a time. Waiting readers and writers must block; they cannot just spin until the lock becomes available.

Note that this Readers-Writer lock specification provides **write priority**. If there are both readers and writers waiting to acquire the lock, a waiting writer must be allowed to acquire the lock before the waiting readers. As a concrete example, if a reader R_1 currently owns the lock, a new writer W_1 must block when it tries to acquire the lock. If reader R_2 arrives while W_1 is blocked and R_1 is still holding the lock, R_2 must also block. When R_1 releases the lock, W_1 must be unblocked and allowed to acquire the lock before R_2.

Implement the following four functions. Structures and global variables can be defined in the provided space.

```
// Define your structures and global variables here.
enum LockTypes {
    READ = 0, WRITE = 1
};
typedef enum LockTypes LockType;

struct rwlock {
    char *lk_name;
    // Add other fields here









};
```

```
struct rwlock *rwlock_create(const char *name)
{
    struct rwlock *lk = kmalloc(sizeof(struct rwlock));
    if (lk == NULL) {
        return NULL;
    }
    lk->lk_name = kstrdup(name);
    if (lk->lk_name == NULL) {
        kfree(lk);
        return NULL;
    }
    // You are allowed to omit error checking code in this function.









}

void rwlock_destroy(struct rwlock *lk)
```

```
void rwlock_acquire(LockType lt, struct rwlock *lk)
```

```
void rwlock_release(LockType lt, struct rwlock *lk)
```

```
struct rwlock {
    char *lk_name;
    // Add other fields here
    struct lock *lk;
    struct cv *cv_read;
    struct cv *cv_write;
    int num_readers_inside;
    int num_writers_waiting;
    bool writer_inside;
};

struct rwlock *rwlock_create(const char *name) {
    struct rwlock *lk = kmalloc(sizeof(struct rwlock));
    if (lk == NULL) {
        return NULL;
    }
    lk->lk_name = kstrdup(name);
    if (lk->lk_name == NULL) {
        kfree(lk);
        return NULL;
    }
    // Omitting error checking
    lk->lk = lock_create(name);
    lk->cv_read = cv_create("Read CV");
    lk->cv_write = cv_create("Write CV");
    lk->num_readers_inside = 0;
    lk->num_writers_waiting = 0;
    lk->writer_inside = false;
    return lk;
}
```

```
void rwlock_acquire(LockType lt, rwlock *lk) {
    lock_acquire(lk->lk);
    if (lt == READ) {
        // Check if there are any waiting writers or if there
        // is a writer inside.
        while (lk->writer_inside || lk->num_writers_waiting > 0) {
            cv_wait(lk->cv_read, lk->lk);
        }
        lk->num_readers_inside++;
    } else {
        while (lk->writer_inside || lk->num_readers_inside > 0) {
            lk->num_writers_waiting++;
            cv_wait(lk->cv_write, lk->lk);
            lk->num_writers_waiting--;
        }
        lk->writer_inside = true;
    }
    lock_release(lk->lk);
}


void rwlock_release(LockType lt, rwlock *lk) {
    lock_acquire(lk->lk);
    if (lt == READ) {
        lk->num_readers_inside--;
        if (lk->num_readers_inside > 0) {
            lock_release(lk->lk);
            return; // Don't need to wake anyone up
        }
    } else {
        lk->writer_inside = false;
    }
    // Wakeup a writer if there are any waiting. Otherwise
    // wakeup all readers.
    if (lk->num_writers_waiting > 0) {
        cv_signal(lk->cv_write, lk->lk);
    } else {
        cv_broadcast(lk->cv_read, lk->lk);
    }
    lock_release(lk->lk);
}

void rwlock_destroy(rwlock *lk) {
    cv_destroy(lk->cv_read);
    cv_destroy(lk->cv_write);
    lock_destroy(lk->lk);
    kfree(lk->lk_name);
    kfree(lk);
}
```