Operating Systems **Mid-Term Exam** ECE344, Fall 2016

Duration: 1 hour 15 min

Examiner: A. Goel

Please fill your student number and name below and then read the instructions below carefully.

Student Number: _____

First Name: _____

Last Name: _____

Instructions

Examination Aids: No examination aids are allowed.

Do not turn this page until you have received the signal to start.

Do not remove any sheets from this test book. Answer all questions in the space provided. No additional sheets are permitted. Use the last blank page as scratch space. This page will not be marked.

This exam consists of 5 questions on 8 pages (including this page). The value of each part of each question is indicated. The total value of all questions is 30.

For the written answers, explain your reasoning clearly. Be as brief and specific as possible. Clear, concise answers will be given higher marks than vague, wordy answers. Marks will be deducted for incorrect statements in an answer. Please write legibly!

Work independently.

MARKING GUIDE

Q1: _____	(5)
Q2: _____	(8)
Q3: _____	(9)
Q4: _____	(6)
Q5: _____	(2)
TOTAL: _____	(30)

Question 1. Labs Are Fun [5 MARKS]

No one wants to hold hot potatoes. The code shown below is invoked by each thread. It tries to move potatoes between participant threads, arranged in a circle. The code is designed to ensure that only *one* thread has a potato at a time. There is a single `potato_lock` global variable.

```
1  int potato[NTHREADS] = {1, 0}; /* potato[0] = 1, potato[i] = 0 when i > 0 */
2
3  void try_move_potato(int num) /* num is the thread number */
4  {
5      assert(interrupts_enabled());
6      lock(potato_lock);
7      if (potato[num]) {
8          potato[num] = 0;
9          potato[(num + 1) % NTHREADS] = 1; /* pass potato to next thread */
10     }
11     unlock(potato_lock);
12 }
```

Part (a) [2 MARKS] Do we need the `lock` and `unlock` code shown above to ensure that only one thread has a potato at a time? If so, show an interleaving that can cause a problem. If not, explain why.

We don't need the lock or unlock in this code. This is because the code first assigns 0 to potato[num], and then increments this value for the next thread. These assignments are atomic because they are writing to a single word. The check in Line 7 ensures that the next thread will pass the potato only after it has the potato. As a result, at no point will multiple threads have a pototo.

Part (b) [3 MARKS] Suppose we switch lines 8 and 9 (shown in bold) in the code above. Now do we need the `lock` code to ensure that only one thread has a potato at a time? Again, explain your answer.

We need the lock and unlock in this case. Consider the interleaving: Thread 1 passes the potato (new line 8), and then Thread 2 runs and passes the potato again (new line 8). At this point, multiple threads have the potato.

Note that even with the lock, Thread 1 and Thread 2 temporarily have the potato, but not other thread knows about it, because the lock ensures mutual exclusion.

Question 2. System Calls [8 MARKS]

When a process invokes the `fork` system call, a child process is created. This process is a copy of the calling process. As we have seen in class, the parent and the child processes run in a unsynchronized manner. In this problem, we will aim to synchronize the two processes. Consider the program listing below. The parent needs to wait until the child has been initialized.

```
int
main()
{
        int pid;
        int child_is_initialized = 0;

        pid = fork();
        if (pid) { /* parent process */
                while (child_is_initialized == 0) { /* spin */
                }
                ...
        } else {   /* child process */
                /* initialize child */
                /* let the parent know that child has been initialized */
                child_is_initialized = 1;
                ...
        }
}
```

Part (a) [1 MARK] Briefly explain why the code above does not work correctly.

The code does not work because the child_is_initialized variable is not shared by the parent and the child after the fork, so any changes to it by the child are not visible to the parent.

Part (b) [1 MARK] Briefly explain why this synchronization cannot be performed without using system calls.

The parent and the child do not share any memory, so there is no way for them to communicate using memory accesses. They need to communicate via the kernel, hence they need to invoke system calls.

Part (c) [4 MARKS] You decide to implement two system calls, `down()` and `up()`, similar to semaphores, to perform the synchronization above. The parent invokes the `down()` call, and the child invokes the `up()` call.

Show what arguments you would pass to these system calls. Make sure to explain why you are passing these arguments. If you need to use new variables in your code, make sure to declare them below (including their types). Assume that you are only allowed to use these two calls, and they implement synchronization correctly in the kernel.

down() is invoked instead of "while (child_is_initialized == 0) {}"

`down(` pid, &child_is_initialized `);`

For a semaphore like operation, we need two things: 1) a counter that has been initialized, 2) a way to ensure that the parent and the child are referring to the same semaphore. We can do both by using "&child_is_initialized". This passes the 0 value as a counter. It also passes the address of child_is_initialized, which will be the same in the parent and the child (due to fork), so the kernel knows that the parent and child are referring to the same object. In addition, we pass the pid value of the child to ensure that the kernel will synchronize this thread with the child process only.

up() is invoked instead of "child_is_initialized = 1;"

`up(` pid, &child_is_initialized `);`

Here the pid value will be 0 (the return value of fork in the child). This will allows the kernel to ensure that this thread will only synchronize with its parent (otherwise, it would synchronize with a process whose process id is pid).

Note that either up() or down() can be called first. When the kernel sees either, it can initialize a new data structure for a semaphore based on seeing a new address (of &child_is_initialized) for a new process, and initialize the counter, based on the value of child_is_initialized. When the second call with the same address occurs, it will ignore the value of child_is_initialized.

Part (d) [2 MARKS] Can you think of a way that the kernel can ensure that another process invoking `down()` or `up()` will not interfere with the synchronization above.

This question was meant to help you with passing the correct arguments above. By passing the pid value in the calls to down(), and up(), the kernel can ensure that a third process (e.g., another child of the parent, or an unrelated process using the same address as the address of child_is_initialized) will not interfere with the synchronization.

Question 3. Synchronization [9 MARKS]

Consider a calculator program that supports arithmetic operations and variable assignments (e.g., `a = 5`, `b = a + 7`). The variable names can only be a single character.

You want to support concurrent computations using condition variables. You want two computations to run concurrently if they don't use any common variables. However, each computation must run atomically. For example, if a computation is using variables `a` and `b`, then other computations using *either* `a` *or* `b` should wait. You make a function `get_vars(const char *expr, char *vars)` that returns the number and the list of all the variables that are used in the `expr` expression. For example, `get_vars(``b = a + 7'', vars)` returns 2, since 2 variables are used. It also sets the first 2 characters of the `vars` array to `a` and `b`.

In the following function, fill in the blank spaces to make your concurrent calculator. All spaces may not need to be filled. You may only use one lock and one condition variable, as declared. You may use `wait(cv, lock)`, `signal(cv, lock)` and `broadcast(cv, lock)`.

```
int symbol_state[256] = { 0 };  /* 256 ascii chars, all elements are 0 */
struct mutex * lock;            /* assume lock and cv are initialized */
struct cv    * cv;
int calculate(const char *expr, int len) { /* assume expr is correct */
    char vars[256];
    int num_vars = get_vars(expr, vars); /* return chars in expr in vars */
    int done = 0, answer, i;

    lock(lock);

    while (!done) {
        done = 1;
        for (i = 0; i < num_vars; ++i)
            if (symbol_state[vars[i]] == 1) {
                wait(cv, lock);
                done = 0;
                break; /* breaks out of for loop, not while loop */
            }
        }
    }
    for (i = 0; i < num_deps; i++)
       symbol_state[vars[i]] = 1;      /* ensure that others wait */

    unlock(lock);

    /* make sure to not run computation within a lock */
    answer = compute_expression(expr, len);

    lock(lock);
    for (i = 0; i < num_vars; ++i)
        symbol_state[vars[i]] = 0;
    broadcast(cv, lock);
    unlock(lock);

    return answer;
}
```

Question 4. Scheduling [6 MARKS]

Consider the 8 threads shown below, with their associated arrival times and processing times:

Job	Arrival Time	Processing Time	Job	Arrival Time	Processing Time
A	0	5	E	7	3
B	2	2	F	8	4
C	3	6	G	9	1
D	5	1	H	11	2

For the two scheduling policies shown below, show the sequence of threads that run on a single processor (if a thread runs for multiple consecutive time units, show it once). When there is a tie, assume FIFO scheduling. For preemptive policies, assume a time slice of *2 time units*. If a job finishes without using up its time slice, scheduling occurs immediately. Also assume that threads arrive just before the time slice expires.

As an example, with FIFO, your output would be:

A	B	C	D	E	F	G	H

Part (a) [3 MARKS] Longest Job First


```
time:   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
arrive: A     B  C     D     E  F  G     H

        A  A  A  A  A  C  C  C  C  C  C  F  F  F  F  E  E  E  B  B  H  H  D  G

Answer: A  C  F  E  B  H  D  G
```

Part (b) [3 MARKS] Shortest Remaining Time

Due to time slice = 2

```
time:   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
arrive: A     B  C     D     E  F  G     H

        A  A  B  B  A  A  A  D  E  E  E  G  H  H  F  F  F  F  C  C  C  C  C  C

Answer: A  B  A  D  E  G  H  F  C

If you ignored the time slice of 2, then we will still accept the answer:

        A  A  B  B  A  D  A  A  E  G  E  E  H  H  F  F  F  F  C  C  C  C  C  C

Answer: A  B  A  D  A  E  G  E  H  F  C
```

Question 5. Labs Really Are Fun [2 MARKS]

You know that implementing thread switching is a little tricky. To simplify matters, you start by implementing thread switching between just two threads, A and B, as shown below. Both have the same code, other than the arguments to the calls to `getcontext`, `setcontext` and the `printf` functions. Assume that the code is run on a single processor.

```
ucontext_t uA, uB;

thread_A() {
    int done = 0;

    while (1) {
        getcontext(&uA);
        if (done == 0) {
            done = 1;
            setcontext(&uB);
        }
        done = 0;
        printf("A makes progress\n");
    }
}
```

```
thread_B() {
    int done = 0;

    while (1) {
        getcontext(&uB);
        if (done == 0) {
            done = 1;
            setcontext(&uA);
        }
        done = 0;
        printf("B makes progress\n");
    }
}
```

You find that when you compile this code normally, it shows the output that you expect: `A makes progress` and `B makes progress` are printed alternately. Now you want to run this code fast, and so you compile this code with compiler optimization options turned on (e.g., `gcc -O thread.c`). When you run the optimized code, you see no output!

When you add some more debugging output in your code, you realize that the context switching is happening. Based on what you have learned about context switching, what might the compiler be doing so that the `printf()` in both the threads shown above does not execute? Assume that the compiler optimizations are generating correct code.

We wanted to test that you understand that registers (but not stack variables) get saved by getcontext.

Without optimization, the "done" variable is allocated on the stack by the compiler.
With optimization turns on, the compiler has allocated a register for the "done" variable (instead of accessing the variable from the stack). On a getcontext, the value of the register (done = 0) is saved in the uA variable, and then changes to this register value (done = 1;) are not reflected in the uA variable. So when setcontext(&uA) happens, done gets the value of 0.

So the context switches keep happening because both threads keep looping in their code.

[Use the space below for rough work.]