Operating Systems **Mid-Term Exam** ECE344, Fall 2015

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 7 pages (including this page). The value of each part of each question is indicated. The total value of all questions is 25.

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: _____ (5)

Q3: _____ (5)

Q4: _____ (5)

Q5: _____ (5)

TOTAL: _____ (25)

Question 1. **All False** [5 MARKS]

Your friend makes the following statements below. You know they are not correct. Clearly explain to your friend **why** they are incorrect. We have provided an example below. Single sentence answers are preferred.

0) On a uni-processor, when a process makes a system call, the system call must return to the process before another process can make a system call.

Answer: A system call can block in the kernel, after which another process can run and issue a system call before the first process's system call returns.

A) When a process invokes a system call, a context switch occurs.

When a process invokes a system call, the OS may return control to the same process, so a context switch will not occur. Some students mentioned that when a process invokes a system call occurs, then a mode switch occurs. This is correct but this explanation doesn't contradict the statement above.

B) A divide-by-zero error in a program invokes the trap instruction.

A divide-by-zero error causes a h/w exception. (a program has to invoke a trap instruction explicitly).

C) A program never executes code after the Unix execve() system call.

A program can execute code located after the execve() system call if the system call returns an error.

D) A Unix wait() system call will always cause a process to block.

A wait() system call will not block if the child process doesn't exist (has exited or wasn't created or on any error).

E) A setcontext() call will always enable signals.

A setcontext call will set the signal state to the state saved by the previous corresponding getcontext call, so it may not enable signals.

Question 2. Thread Switching [5 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. You have added A and B to the run queue, and initialized their thread context so that they will start executing the `thread_A()` and `thread_B()` functions when they are run for the first time. Your scheduler runs Thread A first.

```
int i = 0;
ucontext_t uA, uB;

thread_A() {
    int d = 0;
    while (i < 3) {
        i++;
        printf("A:%d␣", i);
        d = 0;
        getcontext(&uA);
        if (d == 0) {
            d = 1;
            setcontext(&uB);
        }
    }
}
```

```
thread_B() {
    int d = 1;
    while (i < 3) {
        i++;
        printf("B:%d␣", i);
        d = 1;
        getcontext(&uB);
        if (d == 1) {
            d = 0;
            setcontext(&uA);
        }
    }
}
```

Part (a) [3 MARKS] Circle the output you expect to see? Hint: Think carefully about what is saved by the `getcontext()` function.

A) A:1 A:2 A:3

B) A:1 B:2 B:3

C) A:1 B:2 A:3 B:4 A:5 B:6

D) A:1 B:2

E) A:1 B:1

F) A:1 B:2 A:3

We gave 3 marks for a correct answer.

For an incorrect answer, we gave 1 partial mark for Part(a) if Part(b) gave a reasonable explanation for the choice here.

Part (b) [2 MARKS] Briefly explain why you have chosen the answer above.

The main idea that we expected students to understand is that the getcontext call saves the stack pointer and not the values of the stack variables. Execution happens as follows:
print A:1, a:d=0, a:getcontext, a:d=1, a:setcontext, print B:2, b:d=1,b:getcontext,b:d=0,b:setcontext
now we start running a at the point its getcontext was called. The value of a:d is 1 (note that it was set after the getcontext call), so we go to the next iteration of a's loop.
print A:3, a:d=0, a:getcontext, a:d=1,a:setcontext
now we start running b at the point its getcontext was called. The value of b:d is 0, so we go to the next iteration of b's loop. However, i is 3 at this point, so b stops. Thread a nevers runs again.

Question 3. Synchronization [5 MARKS]

Consider the following program that uses three threads (A, B and C). It synchronizes these threads using locks and condition variables. The program begins by invoking the `thread_A()` function.

```
struct lock *l;
struct cv *cv;

void
thread_A()
{
        l = lock_create();
        cv = cv_create();
        thread_create(thread_B, 0);
        thread_create(thread_C, 0);
        thread_yield(THREAD_ANY);
        thread_yield(THREAD_ANY);
        printf("STOP HERE\n");
}
```

```
void
thread_B(void *arg)
{
        lock_acquire(l);
        cv_signal(cv, l);
        thread_yield(THREAD_ANY);
        cv_wait(cv, l);
        lock_release(l);
}

void
thread_C(void *arg)
{
        lock_acquire(l);
        cv_wait(cv, l);
        thread_yield(THREAD_ANY);
        cv_signal(cv, l);
        lock_release(l);
}
```

Part (a) [3 MARKS] Trace the execution of this program until it prints out the message "STOP HERE" by writing down the sequence of context switches that have occurred up to this point. Assume that the scheduler runs threads in FIFO order with no time-slicing (non-preemptive scheduling), all threads have the same priority, and threads are placed in wait queues in FIFO order. The output shown below should be in the form $B \rightarrow A \rightarrow C$, signifying that thread B context switches to thread A, which then context switches to thread C. Hint: Think carefully about how the condition variable primitives work.

A ----------> B ----------> C ----------> A ----------> B ----------> A

yield yield acquire yield wait STOP HERE

Note that when B issues a wait(), it releases the lock, which wakes up C, but C is added to the back of the ready queue after A, so C never gets to run again.

Part (b) [2 MARKS] When the program prints out the message "STOP HERE", list the currently running thread, and the (zero or more) threads in the ready and the wait queues shown below.

current thread: A

ready queue: C

lock wait queue:

cv wait queue: B

Question 4. Waiting for Exit [5 MARKS]

You have implemented an early version of the Unix operating system that provides the signal facility (e.g., the `kill` system call for sending signals). Your operating system also provides the `sleep` system call, which blocks a process until a signal is sent to the process.

Your operating system, however, does not provide the `wait` system call to allow a parent to wait until its child process exits. You consider implementing the functionality of the `wait` system call at the user level with the code shown below. The initial call to `signal` registers an empty signal handler. The `sleep` call blocks until a signal is delivered, after which it returns.

```
void
null(int unused)
{
}

int
main()
{
        int pid;

        signal(SIGUSR1, null);
        pid = fork();
        if (pid) {
                /* block until child sends signal */
                sleep();
                printf("child␣is␣dead\n");
                exit(0);
        } else {
                kill(getppid(), SIGUSR1);
                exit(0);
        }
}
```

Describe two reasons why the code above does not implement the Unix `wait` functionality correctly.

1) 1. Kill can occur before the sleep is issued, so the parent may sleep forever. This is similar to the lost wakeup problem.

2. At the point, "child is dead" is printed, the child may not have exited yet.

3. Parent doesn't get child's exit value.

2) 4. Some other signal sent to the parent will wakeup sleep, even though child hasn't issued kill.

Question 5. **Scheduling** [5 MARKS]

Consider the five threads shown in the table below. Their arrival times and their processing times are known and shown below.

Thread	Arrival Time	Processing Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

A thread scheduler runs these threads in the order shown below. The scheduler has a time slice of 1 time unit. You can assume that threads arrive just before the time slice expires.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Thread	1	1	2	2	3	2	4	4	3	2	4	5	3	2	4	1	5	3	2	4

Part (a) [1 MARK] Circle either a) or b)

a) Scheduler is non-preemptive

b) Scheduler is preemptive

Part (b) [3 MARKS] Briefly describe the policy that you think is being implemented by the scheduler.

longest remaining time first, with round robin scheduling for ties

Some answers suggested highest priority, and assumed that threads were blocking in between. With the assumption that threads are blocking at various points, the scheduler could have been implementing any scheduling policy (e.g., random), and so we did not give marks for this answer.
Some answers suggested longest "job" first, which is a non-preemptive policy. We did not deduct marks in this case.

Part (c) [1 MARK] Describe one problem that can occur with this type of scheduling.

Starve short jobs

[Use the space below for rough work.]