CPS 310 midterm exam #1, 2/19/2016

Your name please: ___________________________  NetID: __________

This exam required you to demonstrate that you understand threads pretty well. It was easy to mess up on the details, particularly on P2 and P3, which ask you to list thread state transitions between [ready, running, blocked] in various scenarios. I saw a lot of students who lost points by leaving key state transitions out, even if they seemed to understand the thread primitives well enough. On P3 and P4 some students did not follow directions well or seemed to be confused by the problem. (Having more experience with threads and thread problems helps.) P5 probes your understanding of P3 and asks you to think about multiple possible schedules at once. A substantial number of students did well on these problems, but many did not.

Partial credit was generous. The short problems were worth 8 points each for P1 and P2, and roughly 4 points each for P5. In general, giving some significant details relating to these questions was worth half credit as long as you didn’t say anything wrong. But in each case I had something specific I was looking for: these problems have short full-credit answers indicated in bold in the solutions. As always, a few students wrote a lot about many things that were tangential to the question. This generally does not help, and it becomes tiresome.

You will see various standard scrawlings on the graded paper. Generally, a check means you got it, a backslash means you didn’t, and a horizontal line is half credit. Plus and minus symbols are quarter credit. Feel free to check my math.

The grade distribution was pretty standard for these midterms (see last page). The high score was a 195. The top 10% or so are in the 170s and 180s. A score of 150 or above is a good exam, putting you on track to earn (let’s say) an A or A- in the class if the labs are good. I appreciate getting these exams and they are strangely fun to grade. Thanks!

A lot of students who have worked hard and are getting by received scores in the 130s or 140s. These are OK but mistakes were made or there are some gaps to be addressed. Scores below 130 correlate with having a bad day or not quite getting it. Scores below 100 can be a serious problem. A few students who seemed to be doing OK were apparently intimidated by P3-P5 and started leaving stuff blank. Sorry! Everybody in this class can do these problems if you work at it a little, breathe, and puzzle through it. But I do understand that time may have been a factor.
Answer all questions. Please attempt to confine your answers to the boxes provided. For the execution tracing problem (P3) you may wish to use scratch paper to keep it clean. There is a synchronization/coding problem at the end: as always: “Any kind of pseudocode is fine as long as its meaning is clear. You may assume standard data structures, e.g., linked lists: don’t write code for those.” If you don’t know the answer to a question, then just say something else relevant for partial credit.

P1. Shorts: kernel and machine (40 points)

(a) What happens when a thread executes code to disable interrupts while it is running in user mode?

(b) Disabling interrupts is often sufficient to synchronize critical sections if a machine has only one processor core, but not if there are multiple cores. Why?

(c) On a system call trap, how does the kernel determine which system call in the kernel API is being requested?

(d) When a thread executes an instruction referencing a virtual address, how does the machine determine which virtual address space to use in translating the address?

(e) How does the kernel determine if a given address translation fault is a “segmentation fault”, i.e., a wild pointer reference?

The machine raises a fault. An instruction to disable interrupts is reserved to kernel mode. The kernel fault handler might kill the process. Half credit for describing what happens if interrupts are disabled successfully. Describing the exception wrong costs 2 points.

Disabling interrupts prevents an involuntary context switch to another thread on this core, but it has no effect on other cores: threads on other cores may enter critical sections that access shared data and race with this thread. Note: such a race may occur even without an interrupt on another core.

The user-mode system call stub writes the system call number (defined by the ABI) into a designated register. After the trap the kernel trap handler reads the value from the register and indexes into a syscall dispatch table, which has a pointer to an internal kernel procedure for the requested syscall.

Each thread is bound to exactly one process, which has exactly one virtual address space (VAS), named by an ID. The VAS ID is part of the thread’s register context; the kernel places the ID in a protected register when it switches into the thread. Details of the ID are machine-dependent. On classic x86 the ID is a pointer to the VAS page table and is stored in control register CR3.

If no translation for an address is present in the page table, then the machine faults and the kernel gains control. The kernel represents the VAS as a set of segments occupying disjoint ranges of virtual addresses. If the address is outside of any segment (“outside the VAS” is OK) then it is a segmentation fault.
P2. Thread primitives and the thread lifecycle (40 points)

As threads execute, they transition through various states (ready, running, blocked). As threads transition among these states, their thread control blocks (TCBs) move among various queues maintained by the thread system. These state changes and queue operations are driven by calls to thread primitives in the thread system API. Below are five thread primitives defined for the lab p1. For each primitive, list the thread state changes it may cause as it executes, and the associated queue operations.

(a) thread_yield(…)
   Caller transitions running→ready and goes to tail of ready queue. [6]
   Another thread from the front of the ready queue runs (ready→running). [2]

(b) thread_lock(…)
   If lock is free then the caller acquires it and continues running.
   Else caller transitions running→blocked and goes to tail of lock queue, and
   another thread from the front of the ready queue runs (ready→running). [2]

(c) thread_unlock(…)
   Caller releases lock and continues running.
   If the lock queue is not empty, then wake up a thread by moving it from the head of
   the lock queue to the tail of the ready queue: it transitions from blocked→ready.

(d) thread_wait(…)
   Caller unlocks the lock (see thread_unlock), blocks (running→blocked), and
   moves to tail of the CV waiter queue. Another thread from the front of the ready
   queue runs (ready→running). Later, when the caller wakes up (e.g., due to
   thread_signal) and runs, it re-acquires the lock (see thread_lock) and returns.
   Forgetting about the lock cost the full credit for this one.

(e) thread_signal(…)
   If the CV waiter queue is not empty, a thread wakes up: it moves from the head of
   the CV waiter queue to the tail of the ready queue (blocked→ready).
Consider the following pseudocode for a soda machine with N=1: the machine has buffer space for a single soda (initially empty). This problem asks you to trace execution of this code in detail for the given main program. Assume that the code runs on a uniprocessor (single core) with no involuntary preemptions: thus the schedule is deterministic. Trace the schedule as an ordered list containing every thread state change. The thread primitives behave as in lab p1: all queues are FIFO, as required for p1t. The threads are named P0 (main), P1, C0, and C1. It may help to list queue contents at various points.

```c
void produce(...) {
    thread_lock(1);
    while (full) {
        thread_yield();
        thread_wait(1, 2);
    }
    add soda;
    thread_signal(1, 2);
    thread_yield();
    thread_unlock(1);
}

void consume(...) {
    thread_lock(1);
    while (empty) {
        thread_yield();
        thread_wait(1, 2);
    }
    take soda;
    thread_signal(1, 2);
    thread_yield();
    thread_unlock(1);
}

main_thread_starts_here() {
    thread_create(consume, ...); /* C0 */
    thread_create(consume, ...); /* C1 */
    thread_create(produce, ...); /* P1 */
    thread_yield();
    produce(); /* P0 */
}
```

P0 runs ...<continue below>
P1 signals → C0 ready
C0 ready
C1 ready
P1 ready
P0 ready (yields)
[ready q: C0 C1 P1 P0]
C0 runs
C0 ready (yields with lock)
C1 runs
C1 blocks on lock
P1 runs
P1 blocks on lock
P0 runs
P0 yields with lock
P0 runs
P0 waits on CV
C0 ready (obtains lock)
C0 ready
C1 runs
C1 blocks on lock
C0 runs
C0 takes soda
C0 signals→C1 ready
C0 yields with lock
C1 runs
C1 blocks on lock
C0 runs
C0 releases lock and exits
C1 runs
C1 waits on CV
[CV q: C0 C1]
P1 ready, P1 obtains lock
P1 runs
Notes on P3 and P5

You could get half credit for showing that you understand how to trace the execution, but if you left a lot of transitions out then you might not get any more than that. If your trace went astray and I was able to figure out what went wrong I tried to give more partial credit. But if I couldn’t figure out what you did I just left it. Some people omitted the transitions and merely listed the queues at each step, but that was long and hard to follow.

It should have been clear that p0 is the main thread, so the first few steps are the C0, C1, and P1 transitioning to ready in calls to thread_create. Many answers omitted these. Thread creation in lab p1 is always asynchronous: the newly created thread does not run until it comes to the front of the ready queue after its creator blocks, yields, or exits and relinquishes the single core.

A common mistake is to allow some other thread to acquire the lock after the holder yields. If a thread yields while holding a lock, then any other thread attempting to acquire that lock must block and wait on the lock queue for the lock holder to run again and release the lock.

In general, if you got the basics right and included enough transitions to follow your trace through to the end, then you might lose only 5 or 10 points for going astray. It was easy to lose track of the ordering in the various queues.

Of course the big trick here is that producers and consumers share a single CV, so it is possible for a signal to go to the “wrong” thread. For example, if a consumer signals another consumer, then the awakened thread absorbs the signal and goes right back to wait; a producer might also be waiting, but does not receive the signal. So it is possible that a producer and a consumer are both waiting for one another, but there is nobody else to signal them. This is a deadlock.

But the deadlock depends on the ordering. And yields can affect the ordering. Consider the first P0 yield. It causes the threads to enter the soda machine in this order: C0, C1, P1, P0. Then C0 and C1 must wait, and P1 awakens C0 to take a soda, but P0 is ahead of C0 in the lock queue, so P0 enters the monitor before C0 takes P1’s soda, and so P0 must wait because the machine is full. When C0 gets in it takes P1’s soda and signals for another producer. But C1 is ahead of P0 in the wait queue, so C1 gets the signal instead of P0 – but C1 just waits again because the machine is empty when it gets there. So C1 is waiting for a producer, and P0 is waiting for a consumer: they are both waiting for each other. But there is nobody else to signal them: deadlock.

But without that first yield the threads enter the monitor in the order P0, C0, C1, P1. Then C0 takes P0’s soda and P1 signals C1 to take P1’s soda, and everyone is happy.

P5 asks you to analyze this situation. To get more than half credit on P5 you had to see that the shared condition variable allows deadlocks to occur, and propose to fix it with broadcasts or separate condition variables.
Part P4. Highway 310 (50 points)

Highway 310 is a two-lane road that passes through a one-lane tunnel. A car can safely enter the tunnel if and only if there are no oncoming cars already in the tunnel. To prevent collisions, sensors at each end of the tunnel notify a controller program when cars arrive or depart. The controller sets a boolean variable \( \text{free} \) and a direction variable \( \text{dir} \) to control signal lights for approaching cars at either end of the tunnel: if \( \text{free} \) is true, then the lights are green in both directions. If \( \text{free} \) is false, then \( \text{dir}=0 \) sets the lights to allow eastbound traffic to pass, and \( \text{dir}=1 \) sets them to allow westbound traffic to pass.

Show how to implement the controller using a monitor. You may assume that each car is represented by a thread that calls \( \text{Arrive()} \) and \( \text{Depart()} \) methods. Threads should block in \( \text{Arrive()} \) if the light is red for their direction, and wake up and continue through when the light is green. \( \text{Arrive()} \) and \( \text{Depart()} \) each take two integer arguments. An eastbound car passes the values 0 and 1. A westbound car passes the values 1 and 0. Your solution should be free of collisions and starvation.

```java
boolean free = true;  int waiters[2] = {0, 0};
int dir = 0;  int cars = 0;

void Arrive (int myway, int otherway) {
    lock();
    while (!free && dir != myway) {
        waiters[myway]++;
        wait(...);
    }
    free = false;
    dir = myway;
    cars++;
    unlock();
}

void Depart (int myway, int otherway) {
    lock();
    cars--;
    if (cars == 0) {
        free = true;
        waiters[myway] = 0;
        waiters[otherway] = 0;
        broadcast(...);
    }
    unlock();
}
```

If you don’t block for waiters in the other direction, then the solution starves. [10]
If you signal instead of broadcast, then you might not get all the waiters for the other direction. [5]
If you don’t let cars into the tunnel when the light is green in their direction, then the solution blocks cars from entering even “if there are no oncoming cars” and no waiters. [10] Holding the lock across arrive and depart is equivalent.
If you don’t keep a count of cars in the tunnel, then the solution is unsafe and may cause collisions. [10]
We took some points for various errors involving getting the direction flips right, waits or signals on the wrong side, etc.
Part 5. Analysis of the P3 trace (20 points)

Answer the following questions about the schedule in Part 3. Also consider possible executions of the “stripped” version, which is the same code, but with the five calls to thread_yield() removed.

(a) Describe the final state of the trace in a single word or sentence.

Deadlock. Note that this is not a “missed wakeup”, since the stuck threads are properly waiting on the CV when “their” signals come. CVs are protected by locks in part to make them invulnerable to missed wakeups.

(b) Is a different outcome possible if the thread_create calls are reordered? (Explain.)

Yes. In the project p1 scenario the threads execute in the order they are created. If threads enter the monitor in a different order the deadlock may be avoided. Answers that analyzed invariants were appreciated, e.g., “no, because every consumer requires its own producer to allow it to pass the wait”. But to get this right you really had to see the potential for deadlock.

(c) Does the “stripped” version have a different outcome, i.e., do the calls to thread_yield affect the result?

Once all threads are queued at the monitor, there is only one thread ready at a time...except for the brief window after a signal, when the awakened thread runs just long enough to queue at the lock. So some yields don’t matter. But it matters when P0 yields in main() it allows the other threads to run ahead of it. Without that yield, the order is P0, C0, C1, P1, which completes.

(d) What outcomes are possible in the “stripped” version if it runs with preemption enabled? (Explain.)

Preemption essentially puts yields anywhere, randomly, so either outcome is possible: it might deadlock or it might complete successfully. Because the locking is correct no other outcomes are possible, e.g., it is not possible to consume the same soda twice or anything else weird like that.

(e) What changes would you make to the “stripped” code before deploying it in production?

This question asks directly if you saw the error. The correct fix is to use two CVs instead of one (as in the soda machine solution in class), or broadcast instead of signal.
Here is the grade distribution. People always ask for it, and it always looks like this.