

CSE 120 Principles of Computer Operating Systems
Fall Quarter, 2002
Halloween Midterm Exam

Instructor: Geoffrey M. Voelker

Name _____

Student ID _____

Attention: This exam has six questions worth a total of 70 points. You have approx. 80 minutes to complete the questions. As with any exam, you should read through the questions first and start with those that you are most comfortable with. If you believe that you cannot answer a question without making some assumptions, state those assumptions in your answer.

1	/4
2	/15
3	/6
4	/15
5	/15
6	/15
Total	/70

1. (4 pts) Potpourri: Answer yes or no, or with a single term or short answer, as appropriate. You should go through these quickly, answering with the first answer that comes to mind — it probably is the correct one. Only dwell on ones you are unsure of after finishing the other questions.
 - (a) A resource allocation graph that has a cycle in it always/sometimes/never means that there is a deadlock?
sometimes (see the example in deadlock lecture)
 - (b) Can the system call instruction be a privileged instruction?
no – if it is, then user programs cannot invoke kernel operations
 - (c) In what Nachos file was the Semaphore class implemented?
synch.cc
 - (d) Round-robin scheduling always/sometimes/never results in more context switches than FCFS?
sometimes – if every job has an execution time less than the quantum, then it has the same number as FCFS.
some answers began “always, except...” – remember that the golden rule with always/sometimes/never questions is that “always, except” means sometimes.

2. (15 pts) Give brief answers to each of the following questions.

- (a) What is the purpose of system call instructions, and how do they work? Give two examples of common system calls.

System call instructions enable user programs to invoke operations in the kernel. System call instructions cause an exception, which causes the CPU to vector into the OS. The OS saves the process state, detects that it is a system call exception, identifies the system call number from the argument to the instruction, and invokes the system call routine. When the routine finishes, the OS restores the process state and returns from the system call.

Two common examples are read() and write().

- (b) What is the difference between interrupts and exceptions? Give two examples of each.

Interrupts are asynchronous events external the CPU (e.g., timer interrupt, device interrupt). Exceptions are synchronous events that occur as the result of executing instructions (e.g., divide by zero, system call).

- (c) What is the difference between fork() and exec() on Unix?

fork() creates a new process and copies the address space from the parent into the new child process. exec() stops executing the program in the process, overwrites it with a new program, and starts executing the program at the beginning; exec() does not create a new process.

- (d) What is the difference between a race condition and deadlock?

A race condition is caused by more than one thread accessing shared state without synchronization. Threads continue to execute, but the program can have incorrect results. In a deadlock, threads cannot make any progress. They are all waiting to acquire resources that each other holds (circular wait) while the three other deadlock conditions hold.

- (e) What is the difference between Thread::Yield() and Thread::Sleep()?

Yield() places the current thread on the end of the ready list, and context switches to another thread. Sleep() halts the current thread and context switches to another thread. In particular, Sleep() does not put a thread on a wait list (other code has to do that since Sleep() does not know which list the thread should wait on).

3. (6 pts) Round-robin schedulers (e.g., the Nachos scheduler) maintain a *ready list* or *run queue* of all runnable threads (or processes), with each thread listed at most once in the list. What can happen if a thread is listed twice in the list? Briefly explain how this could cause programs that use synchronization primitives to break on a uniprocessor.

The problem is that a thread on the ready list twice results in a spurious wakeup. Consider a thread executing the PingPong procedure in the lecture notes. If a thread is on the ready list twice, then the first time it runs on the CPU it will execute inside of PingPong and eventually Wait on the condition variable. But, since it is on the ready list twice, it will run again, without any other thread having called Signal on the condition variable. This could, for example, cause that thread to print twice in a row, violating the problem constraints.

The most common error on this problem was to think of the thread on the list twice as two different threads, with two different stacks, etc. Although this is not accurate, I still gave partial credit when an answer with this interpretation demonstrated a synchronization problem.

4. (15 pts) The table below lists five jobs, the time that they arrive in the system (when they are created), how long they take to execute, and their priority (higher values correspond to higher priorities). For example, job *A* arrives at time “10”, requires “30” units of execution time to complete, and has priority “1”.

Job	Arrival Time	Execution Time	Priority
A	10	30	1
B	20	50	5
C	30	10	3
D	40	40	2
E	50	20	0

For all jobs in each of the following scheduling algorithms, calculate (a) the **Start Time**, the time at which the job is first scheduled to run (it may have to wait when it arrives), and (b) the **End Time**, the time when it finishes executing.

(a) **First Come First Serve (FCFS)**

Job	Start Time	End Time
A	10	40
B	40	90
C	90	100
D	100	140
E	140	160

(b) **Shortest Job First (SJF)**

Job	Start Time	End Time	Job	Start Time	End Time
A	10	50	A	10	40
B	110	160	B	110	160
C	30	40	(SRTF) C	40	50
D	70	110	D	70	110
E	50	70	E	50	70

(c) **Priority**

Job	Start Time	End Time
A	10	140
B	20	70
C	70	80
D	80	120
E	140	160

5. (15 pts) Consider the following test program for an implementation of locks and condition variables in Nachos. It begins when the “main” Nachos thread calls ThreadTest(). Trace the execution of this program until it prints out the message “STOP HERE” and (a) write down the sequence of context switches that occurred up this point, (b) the output of the program, and (c) list the queues that the threads are on at this point, and their relative order if more than one thread is on a queue (currentThread, the readyList, and any wait queues associated with synchronization primitives). For example, “A -> B” signifies that thread A context switches to thread B, and “readyList: A” signifies that A is on the ready list.

Assume that the scheduler runs threads in FIFO order with no preemptive time-slicing (non-preemptive scheduling), all threads have the same priority, and threads are placed on wait queues in FIFO order.

```

Lock *pumpkin;
Condition *cv;

void A(int arg) {
    pumpkin->Acquire();
    printf(`giddy`);
    cv->Wait(pumpkin);
    printf(`gaddy`);
    currentThread->Yield();
    cv->Signal(pumpkin);
    pumpkin->Release();
}

void B(int arg) {
    pumpkin->Acquire();
    printf(`goody`);
    cv->Signal(pumpkin);
    currentThread->Yield();
    printf(`geddy`);
    cv->Wait(pumpkin);
    pumpkin->Release();
}

void ThreadTest() {
    Thread *t;

    pumpkin = new Lock("l");
    cv = new Condition("cv");

    t = new Thread("A");
    t->Fork(A, 0);
    currentThread->Yield();
    t = new Thread("B");
    t->Fork(B, 0);
    currentThread->Yield();
    currentThread->Yield();
    printf(`STOP HERE\n`);
}

```

(a) Context switches: *main -> A -> main -> B -> main -> A -> B -> main*

(b) Output: *giddy goody geddy*

(c) Thread queues at “STOP HERE”:

```

currentThread: main
readyList: A
pumpkin:
    cv: B

```

The one subtlety to this problem is at the end. When A is woken up from waiting on the condition variable, it does not immediately return. Instead, it first tries to reacquire the lock (that is held by B), gets placed on the lock’s wait queue, and finally placed on the ready list when B finally calls wait (but A never does run again).

6. (15 pts) Because his house is so popular on Halloween, Greg Ghoul likes to hand out Halloween candy in the following way. Each trick-or-treater coming to his door takes a single “ticket” with a number from a ticket machine on his porch. The ticket numbers dispensed are guaranteed to be unique and sequentially increasing. When Greg is ready to give candy to the next trick-or-treater, he calls out the “candycount”, the lowest unserved number previously dispensed from the ticket machine. Trick-or-treaters wait until the candycount reaches the number on their tickets. Greg waits until the trick-or-treater with the ticket he called comes forward with their bag held open before giving out candy.

Show how to implement `Greg()` and `trick-or-treater()` procedures to solve this problem using locks and condition variables. Represent calling out the “candycount” by printing out its value in the appropriate place in `Greg()`. In addition, print out a message in `trick-or-treater()` when the trick-or-treater gets candy and print out a message in `Greg()` when Greg gives out candy.

```
// shared variables
```

```
Lock mutex;
Condition candy, greg;
int candycount = 0, ticketcount = 0;
```

```
void Greg() {
    mutex->Acquire();
    while (1) {
        printf("Candy for %d!\n",
              candycount);
        candycount++;
        candy->Broadcast(&mutex);
        greg->wait(&mutex);
        printf("Gave away the candy!\n");
    }
    mutex->Release();
}
```

```
void trick-or-treater() {
    int myTicket;

    mutex->Acquire(&mutex);
    myTicket = ticketcount++;
    while (myTicket != candycount) {
        candy->Wait(&mutex);
    }
    printf("Treater %d got candy!\n",
          myTicket);
    greg->Signal(&mutex);
    mutex->Release();
}
```