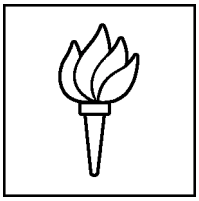


Computer Science Department

New York University

G22.2250-001 Operating Systems: Spring 2009

Midterm

All problems are open-ended questions. In order to receive credit you must answer the question *as precisely as possible*. You have 80 minutes to answer this quiz.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

I (xx/16)	II (xx/10)	III (xx/10)	IV (xx/10)	V (xx/14)	VI (xx/10)	Total (xx/70)

Range of scores	number of students
< 30	1
30-39	3
40-49	5
50-54	8
55-59	5
> 60	4

I Basic OS knowledge

Answer the following multiple-choice questions. Circle *all* answers that apply. Each problem is worth 4 points. Each missing or wrong answer costs -2 point.

A. In a typical modern OS, which of the following statements are true about user applications and the Kernel?

1. Application may directly invoke any function calls in the Kernel.
2. The Kernel executes an application's machine instructions the way a JVM executes Java bytecode.
3. Applications run at superviser privilege level (i.e. CPL=0).
4. The Kernel runs at superviser privilege level (i.e. CPL=0).

Ans: 4 only. Some also included 2 which is not correct. JVM has a runtime system that decodes and executes the bytecode. A typical kernel directly lets CPU execute the application's machine instructions.

B. Which of the following instructions should be protected, i.e., can execute only when the processor is running at superviser level?

1. OUTB (contact I/O device).
2. SUB; (subtract numbers)
3. CLI; (disable interrupt)
4. JMP; (jump to a different instruction)
5. INT; (invoke an exception, e.g. for jumping to kernel's syscall dispatcher with kernel privilege)

Ans: 1 and 3 only. INT is allowed because that's what applications use for invoking syscalls. JMP is allowed because it simply jumps the execution to a different point in the program code (which are commonly used to implement loops and conditional branches).

C. Which of the following are elements of a typical process descriptor? (Recall that a process descriptor or process control block is the per-process state kept by the Kernel.)

1. Disk driver;
2. File descriptor table.
3. The saved register values.
4. Process state (blocked/runnable);

Ans: 2,3,4.

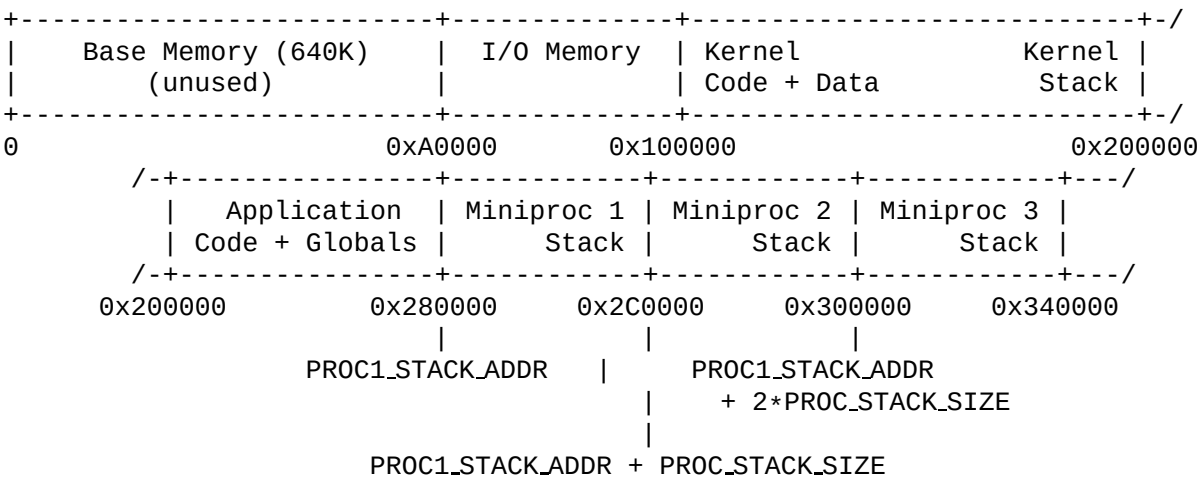
D. Which of the following statements are true when the kernel switches execution from the currently running process to another runnable process?

1. The kernel *must* save the register values (including %eip,%esp etc.) of the current process to memory.
2. The kernel *must* close all the files opened by the current process.
3. The kernel *must* restore the register values of the next process to be executed.
4. The kernel *must* save the content of the current process's memory on disk.

Ans: 1.3 only

II Kernel basics

Recall that Lab1's MPOS kernel has the following memory map.



1. [5 points]: Ben Bitdiddle notices that MPOS has only a single kernel stack for all mini-processses. Ben also reads that Linux uses a separate kernel stack for each process. Explain the advantages for using a separate kernel stack per process on a uni-processor machine. (Hint: Can multiple mini-processes simultaneously be active in the kernel portion of the code with a single kernel stack?)

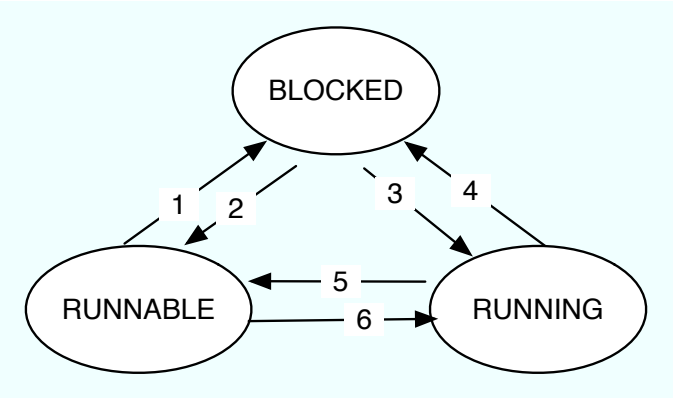
Ans: With a single kernel stack, only one kernel path can be using the stack at any given time. Therefore, when using a single kernel stack, no kernel functions can block in the middle since a blocking function will hog the stack, preventing other kernel paths from executing. In a design that keeps a separate kernel stack for each process, one has the flexibility to block a kernel path in the middle. For example, if the kernel path needs to read from disk in the middle of a function, it's convenient to block the currently executing kernel context and schedule some other kernel/user threads.

2. [5 points]: Ben wants to make the MPOS kernel work on a multi-core machine. In particular, he wants multiple processors to be able to simultaneously execute in some kernel code paths. Can he have a single kernel stack as before? Explain. If your answer is no, also explain what the minimum number of required kernel stacks is.

Ans: The minimum number of required kernel stacks is equal to the number of cores if Ben wants multiple processors to be able to simultaneously execute some kernel code paths, since no two active code paths can share the same stack.

III Scheduling

3. [10 points]: A (mini-)process or thread moves between each of the three states during its lifetime. Describe conditions that cause a thread to go through each of the 6 state transitions. Be specific. If there are more than one conditions that might cause the transition, list at least two of them. Label it N/A if the corresponding transition never occurs.



Ans: 1: N/A
2: A blocked process becomes runnable if the resource it is waiting for becomes available. This could happen if the lock it is attempting to acquire becomes free or the disk block it is attempting to read has been fetched to memory.
3: N/A.
4: A running process becomes blocked if the resource it is trying to obtain is temporarily unavailable. For example, it is attempting to acquire a mutex held by other processes or it is trying to read a disk block.
5: A running process becomes runnable when it gets preempted due to a timer interrupt.
6: A runnable process becomes running when the kernel decides to execute it.

IV Synchronization

After consulting the Intel manual, Ben Bitdiddle finds out that x86 supports additional atomic instructions other than test-n-set and compare-n-swap. In particular, there exists an atomic ADD/SUB instruction (in C equivalent, `atomic_add(int* v, int i)` or `atomic_sub(int* v, int i)`) that atomically adds/substracts a value to a memory location (i.e. `*v += i` or `*v -= i`). Ben sets out to re-write the spin lock funtions as follows:

```
struct Lock {
  int locked; //0 means unlocked, positive values mean locked
}

void acquire(Lock *l)
{
  while (1) {
    atomic_add(&l->locked, 1);

    //atomic_read acts as a memory barrier to ensure
    //sequential consistency of momory operations.
    if (atomic_read(&l->locked) == 1) {
      break;
    }
    l->locked--;
  }
}

void
release(Lock *l)
{
  l->locked--;
}
```

4. [10 points]: Does Ben's new spin-lock implementation work? If yes, explain your reasoning. If not, give a concrete scenario for when it fails and write down your fix by directly modifying the code above.

Ans: Ben's new spin-lock does not work. The main problem is that `l->locked--` *in the* `acquire` *and* `release` *functions is not atomic, causing races. Here's an example race. The lock value is 1 and thread t1 is holding the lock. Thread t1 releases the lock at the same time thread t2 is trying to acquire the lock. Thread t1 and t2 end up executing the* `l->locked--` *statements in* `acquire` *and* `release` *functions concurrently. Both reads the* `l->locked` *variable into registers with value 2 and both writes to the* `l->locked` *variable with value 1. Subsequently, no thread is holding the lock (as t1 has finished releasing the lock) and yet the lock value is 1, preventing all threads from ever acquiring the lock again. The fix is to subsititue both* `l->locked--` *statements with* `atomic_sub(&l->locked, 1)`.

Many of you have also identitied another potential problem with the fixed implementation. In particular, it potentially suffers from livelock where multiple threads concurrently execute `acquire` *when the lock is free and yet none of the threads is able to successfully acquire the lock after many tries due to unfortunate timing.*

V Synchronizing the Kernel

Ben Bitdiddle decides to modify his Lab 1 MPOS to run on a multi-core machine. Since multiple cores might be executing in the kernel code, Ben uses a big kernel lock (`big_lock`) to prevent races on shared data structures, such as the array of process descriptors (`proc_array`). Ben uses an array (with size `NUM_CPUS`) of process pointers to keep track of the current running process on different CPUs. The function `cpu()` returns the identifier of the CPU (from `0` to `NUM_CPUS-1`) that is executing the code. Below is his modified `interrupt` function for handling syscalls.

```
...
process_t proc_array[NPROCS];
process_t* current[NUM_CPUS];
...


void
interrupt(register_t *reg) {
  current[cpu()]->p_registers = *reg;
  current[cpu()]->p_state = P_RUNNABLE;

  switch (reg->reg_intno) {
    case INT_SYS_GETPID:
      current[cpu()]->p_registers.reg_eax = current[cpu()]->p_pid;
      schedule();
    case INT_SYS_YIELD:
      schedule();
    default: /* Ben only implemented two syscalls*/
      panic();
  }

}

void
schedule(void)
{
  pid_t pid = current[cpu()]->p_pid;
  while (1) {
    pid = (pid + 1) % NPROCS;
    acquire(&big_lock);
    if (proc_array[pid].p_state == P_RUNNABLE) {
      current[cpu()] = &proc_array[pid];
      current[cpu()]->p_state = P_RUNNING;
      release(&big_lock);
      /* Recall that the run function in Lab 1 restores saved register values
         and returns to the user context of the process */
      run(current[cpu()]);
    }
    release(&big_lock);
  }
}
```

5. [4 points]: As seen in the code above, Ben has introduced a new process state `P_RUNNING` that was not present in the MPOS kernel of Lab 1. Explain why this new state is necessary.

Ans: The new state `P_RUNNING` *is necessary to prevent multiple CPUs from executing the same runnable process.*

6. [10 points]: Is Ben's code correct? Explain. If you don't think his modification is correct, show your fix by directly modifying his code.

Ans: No. Ben's code suffers from a race, similar to the sleep-wakeup race discussed in the class. Suppose a mini-process T1 running on CPU 0 invokes the getpid() syscall. As a result, CPU 0 executes the code in `interrupt` *and sets the process state from* `P_RUNNING` *to* `P_RUNNABLE` *(in the second line of* `interrupt()`*). As the same time, CPU 1 attempts to schedule a process by executing the* `schedule()` *function. It grabs the* `big_lock` *and sees that T1 is runnable and hence proceeds to run T1. Since CPU 0 has not finished storing the syscall return value in T1's saved registers, T1 will see an incorrect getpid() result.*

One possible fix is to insert `acquire(&big_lock)` *as the first line in* `interrupt()` *and insert two* `release(&big_lock)` *before calling schedule() in the* `interrupt()` *function. This ensures that when one CPU is in the midst of modifying a process's state (including its running status), no other CPUs can execute the process.*

VI G22.2250-001

7. [10 points]: We would like to hear your opinions about the class so far, so please tell us what you like the most/least about this class. Any suggestions for improving the class are welcome.

End of Midterm