UCLA Computer Science 111 (Fall 2008)
Midterm
100 minutes total, open book, open notes

Name:_____ Student ID:_____

1	2	3	4	5	6	7	8	9	total

1 (8 minutes). Suppose the x86 architecture did not trap when executing privileged instructions in non-privileged mode. Instead, when the CPU is in normal (non-privileged) mode and executes a privileged instruction such as INT, nothing happens; it is equivalent to a no-op. If this were true, could you implement system calls to enforce hard modularity for an operating system like Linux, with acceptable performance? If so, explain how; if not, explain why not.

2 (6 minutes). Suppose you run the following shell command in an ordinary empty directory that you have all permissions to. What will happen? Describe a possible sequence of events.

```
cat <mouse | cat | cat >dog
```

3 (10 minutes). WeensyOS 1, like many operating systems, implements getpid as a system call. Would it be wise to implement it as an ordinary user-space function whose implementation simply accesses static memory accessible to the current process? Explain the pros and cons of this alternate implementation.

4. POSIX requires that a 'write' system call requesting a write of fewer than PIPE_BUF bytes to a pipe must be atomic, in the sense that if multiple writers are writing to the same pipe, the output data from the 'write' cannot be interleaved in the pipe with data from other writers. PIPE_BUF is up to the implementation, but must be at least 512 (on Linux, it's 4096). A 'write' of more than PIPE_BUF bytes need not be atomic.

4a (5 minutes). What's the point of this atomicity requirement? What sort of program works if the implementation satisfies this requirement, but does not work otherwise?

4b (5 minutes). Suppose you want to write a super-duper OS in which PIPE_BUF is effectively infinity (it's equal to 2**64 - 1, say). What sort of problems do you anticipate having with your OS?

4c (5 minutes). Did your solution to Lab 1b rely on this requirement? If so, explain where; if not, give an example of what happens if the requirement is not met.

5 (8 minutes). One way that a process can notify another is via the 'kill' system call. Another way is via the 'write' system call, via a pipe. Isn't it redundant to have two different ways to send notifications? Wouldn't it be simpler if the API omitted 'kill', and we asked programmers to use only 'write' to send notifications? If so, explain how you'd replace arbitary calls to 'kill' with calls to 'pipe' followed by calls to 'write'; if not, explain why 'kill' cannot in general be replaced in this way.

6 (8 minutes). Does it make sense to use SJF in a preemptive scheduler? If so, give an example; if not, explain why it doesn't make sense.

7 (8 minutes). Suppose you have a single blocking mutex (not a simple mutex) around a shared pipe object. Suppose each reader and writer locks the object only for a short period of time, and that there are plenty of active readers and writers. Can a would-be reader starve? If so, show how. If not, explain why not. If your answer depends on the implementation, explain your assumptions and why they matter.

8. Consider the following source code, adapted from the implementation of sys_wait in mpos-kern.c's 'interrupt' function, in WeensyOS 1.

```
 1  void
 2  interrupt(registers_t *reg)
 3  {
 4    current->p_registers = *reg;
 5    switch (reg->reg_intno) {
 6      ...
 7    case INT_SYS_WAIT: {
 8      pid_t p = current->p_registers.reg_eax;
 9      if (p <= 0
10          || p >= NPROCS
11          || p == current->p_pid
12          || miniproc[p].p_state == P_EMPTY
13          || 0)
14        current->p_registers.reg_eax = -1;
15      else if (0
16               || miniproc[p].p_state == P_ZOMBIE
17               || 0)
18        current->p_registers.reg_eax = miniproc[p].p_
exit_status;
19      else
20        current->p_registers.reg_eax = WAIT_TRYAGAIN;
21      schedule();
22    }
23
24    default:
25      for (;;)
26        continue;
27    }
28  }
```

For each of the following lines in the source code, give an example of exactly what could go wrong, from the application's viewpoint, if you omitted that particular line:

8a (3 minutes). Line 4
8b (3 minutes). Line 10
8c (3 minutes). Line 11
8d (3 minutes). Line 12
8e (3 minutes). Line 16
8f (3 minutes). Lines 19 and 20 (omitting both at once)
8g (3 minutes). Line 21 (replacing it with "break;")

9. Suppose we have an implementation of pipes in which we write not just single bytes, but large objects all at once. Our idea is to have "super fine-grained locking", one in which there is a lock associated with each object in the buffer. (This would be consume a lot of memory for our previous single-byte implementation, of course, since the locks are much bigger than single bytes; but for big objects the memory overhead is negligible.) The hope is that with super fine-grained locking, we can have even more simultaneous threads access our pipes than we could with ordinary fine-grained locking. Here's the implementation:

```
enum { N = 1024, LARGE = 1048576 };

typedef struct { char contents[LARGE]; } large_object;

struct pipe {
  mutex_t bufm[N];
  large_object buf[N];
  mutex_t rm, wm;
  size_t r, w;
};

void writeobj(struct pipe *p, large_object *o) {
  lock(&p->wm);
  while (p->w - p->r == N)
    continue;
  size_t w_mod_N = p->w++ % N;
  lock(&p->bufm[w_mod_N]);
  p->buf[w_mod_N] = *o;
  unlock(&p->bufm[w_mod_N]);
  unlock(&p->wm);
}
```

9a (8 minutes). This implementation has some correctness bugs. Identify and fix them. Do not fix performance bugs, just correctness ones.

9b (8 minutes). Now, let's take a look at the performance bugs. Assuming the correctness bugs are fixed, how well does this implementation achieve the performance goals for super fine-grained locking as described above? For each performance flaw you find, suggest how you'd go about improving it, if the goal is mainly to increase parallelism. Or, if it can't be improved, explain why not.