UCLA CS 111 Midterm, Spring 2010
100 minutes total, open book, open notes
Name: Student ID:

1	2	3	4	5	total

1. (12 minutes). The program that we wrote for our paranoid English professor did I/O by calling a subroutine instead of by using system calls. Suppose we wanted to modify it to use system calls, making a relatively small set of changes so that these system calls would trap into a tiny kernel that did the I/O. Would this also solve the professor's problem? If so, describe what changes would need to be made, and the pros and cons of this change; if not, explain why not.

2. (16 minutes). Suppose we are on a version of Linux in which the pipe() system call has been removed and does not work. However, we still want to implement a shell that supports syntax like "CMD1 | CMD2". We decide to implement it using "ersatz pipes", as follows:

* The parent shell creates a temporary file, /tmp/pipe3196 say, where the "3196" is chosen so as not to collide with other file names.
* The parent shell opens the file with read-write access.
* The parent shell removes the file.
* The parent shell forks and execs CMD1 with standard output being the file
* Likewise for CMD2 exit and standard input.
* When CMD1 and CMD2 exit, the kernel automatically removes the file

List the drawbacks of ersatz versus true pipes, in decreasing order of practical importance.

3. Consider the following code:

```
1    int sortIO(void) {
2         pid_t p = fork();
3         switch (p) {
4              case -1:
5                   return -1;
6              case 0:
7                   execvp("/usr/bin/sort",
8                        (char *[]) {"sort", NULL});
9                   _exit(1);
10                  break;
11             default: {
12                  int status;
13                  if (waitpid(p, &status, 0) < 0)
14                       return -1;
15                  if (!WIFEXITED(status)
16                            || WEXITSTATUS(status) != 0)
17                       return -1;
18                  return 0;
19             }
20        }
21   }
```

For each of the following proposed changes, explain the consequences of the change. Consider each change independently.

3a (3 minutes): Omit line 10.
3b (3 minutes): Omit line 4 and 5.
3c (3 minutes): Omit line 5.
3d (3 minutes): replace lines 9 and 10 with “return -1;”.
3e (3 minutes): Interchange the labels “case 0:” with “return -1;”.
3f (3 minutes): Replace line 17 with “{errno = EIO; return -1;}”.
3g (3 minutes): Change line 8’s “sort” to “cat”.

4. Consider the following, discussed in class:

```
struct rlocl{
     lock_t L;
     unsigned int readers;
};

void lock_for_reading (struct rlock *p) {
     lock(&p->L);
     p->readers++;
     unlock(&p->L);
}

void unlock_for_reading (struct rlock *p) {
     lock(&p->L);
     p->readers--;
     unlock(&p->L);
}

void lock_for_writing (struct rlock *p) {
     for (;;) {
          lock(&p->L);
          if (&p->readers)
               break;
          unlock(&p->L);
     }
}

void unlock_for_writing (struct rlock *p) {
     unlock(&p->L);
}
```

4a (5 minutes). Would it be cleaner to add a “const” to the type of the argument to lock_for_reading and unlock_for_reading, so that it is “struct rlock const *”? the idea is that the caller needs only read access, and “const” is C’s way of saying “I need just read access”.

4b (6 minutes). Suppose this code is running under Linux, with multiple threads accessing the locks in question, and suppose that we are running on a single-core CPU. Suppose thread 1 has a write lock but will give it up after it executes a few more instructions, but suppose that thread 2

is currently running and attempts to gain a read lock. Explain what will happen next, assuming that no other threads are accessing the lock.

4c (6 minutes). An application has "superlinear speedup" if it runs more than N times faster on a machine with N CPUs than it runs on a machine with a single CPU. Explain how a multithreaded Linux application that uses a struct rlock variable heavily might have superlinear speedup.

4d (10 minutes). Suppose your multithreaded Linux application crashes and you have a core dump saving the entire state of your process. You are concerned about a particular struct rlock variable named RL. How can you tell which threads held a read lock on RL at the point of the crash, and which held a write lock on RL?

4e (6 minutes). This code has a bug when the reader counter overflows. Fix the bug, by assuming that any lock with that many readers will forever have readers, and that writers should therefore be locked out forever after. Do no modify "struct rlock" in your fix; that is, modify only the code, not the data layout.

5 (15 minutes). Suppose that, instead of _preventing_ race conditions, we want to _encourage_ race conditions. That is, we want to lock out all accesses to an object _except_ when there is one writer thread and at least one other accessing thread. (The idea is that we want to have an object whose contents are as unreliable as possible, so that the object can be used as part of an entropy pool for a pseudorandom number generator. OK, so this is a crackpot idea, but anyway that's the goal.)

Modify the code in question 4 so that it encourages race conditions as described above, instead of preventing them. Start with the original question-4 code; do not fix the overflow bug mentioned in question 4e. You may modify the declaration of "struct rlock".