Midterm 1–February 23, 1990

Open book section (36 points)

The exam is to be turned in at 2:50 PM. The closed book section should be turned in before you open your books and notes to work the open book section. For the open book section, write your answers either on the exam in the space provided (if you can write very small) or on separate pieces of paper.

I have neither received nor given any unauthorized aid on this exam. ________________

There is a Unix utility `pwd` which prints the pathname of the current directory, as in:

```
% pwd
/usr/vsr/wsr
```

`pwd` uses a tree ascent algorithm to accomplish its task. It starts at the current directory and follows the ".." pointers to the root directory. At each step of the recursive ascent, `pwd` has a simple task. It has a reference to a directory D as a relative pathname. It must search the parent directory of D to find a component name referring back to D. In our example, if D is "`../../`" (*i.e.,* "`/usr/`"), the component name to be discovered is `vsr`. Consequently, each recursive step of `pwd` begins with the search:

```
foreach component $C$ of $D$/..
    if $C$ refers to $D$ then break ;
```

This question examines the complexities of determining when C refers to D. The name of a component, C, of D`/..` is obtained by searching the bytes of D`/..`'s directory file. The only other relevant information about C contained in the directory file is its inode number. We will explore how C's name and inode number *as stored in the directory* can be used to determine if C refers to D. (Forget everything you know about symbolic links before you write your answer. They're irrelevant in this context.)

Problem 1. (4 points)

It is possible that C's inode number *as stored in the directory* is not the same as the *real* inode number used to access C in the inode table. When can C have two different inode numbers?

Problem 2. (4 points)

Furthermore, it is possible that C and D may have the same *real* inode numbers even though they refer to different directories. Give an example illustrating this.

Problem 3. (9 points)

Thus, in general `pwd` must obtain a copy of the *real* in-core inode of each component C of the parent directory to see if C really refers to D. `pwd` uses the file status system calls [§5.11] to obtain inode copies. What disk accesses must the kernel make in order to get the inode? Is this likely to be expensive in terms of machine resources? Will the last inode obtained "cost" as much as the first?

Problem 4. (6 points)

Assume that we really want to avoid these extra disk accesses whenever possible. Use the facts you've uncovered in the last three questions to construct an efficient test for determining when C refers to D. That is, figure out when you can avoid obtaining a copy of C's inode.

Problem 5. (6 points)

Could you speed up `pwd` by having it test if C is a directory before getting C's inode. If so, how? If not, why not?

Problem 6. (6 points)

Could you speed up `pwd` by having it read the mount table? If so, how? If not, why not?

Problem 7. (1 point)

Which algorithm of the Unix kernel could loosely be described as the "inverse" of `pwd`. (If you can't remember the name, just give a two sentence description. Don't bother looking it up.)