CSCI 493.66 Unix Tools

with emphasis on string algorithms and Bioinformatics

Homework 4
Due 04/02/09

Saad Mneimneh
Computer Science
Hunter College of CUNY

Problem 1: Fixing pwd

The following is an adaptation of the C code for pwd provided in the book, in C++ and Perl respectively:

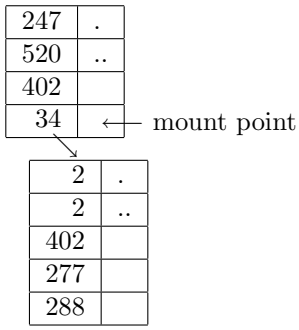
```
#include <iostream>
#include <string>
#include <sys/stat.h>
#include <dirent.h>

using std::cout;
using std::string;

ino_t name2inode(const char * name) {
  struct stat info;
  stat(name, &info);
  return info.st_ino;
}

string inode2name(ino_t inode) {
  int dircount=0;
  DIR * dirp=opendir(".");
  dirent * direntp;
  struct stat info;
  while (direntp=readdir(dirp)) {
    stat(direntp->d_name, &info);
    if (direntp->d_ino==inode)
      return string(direntp->d_name);
  }
  return string();
}
```

```
void pwd(ino_t inode) {
  string name;
  struct stat info;
  stat("..", &info);
  if (info.st_ino != inode) {
    chdir("..");
    name=inode2name(inode);
    pwd(info.st_ino);
    cout<<"/"<<name;
  }
}

int main() {
  pwd(name2inode("."));
  cout<<"\n";
}
```

```
#!/usr/bin/perl -w
use strict;

sub name2inode {
  my $name=shift;
  (my $dev, my $ino,my $mode,my $nlink,
   my $uid,my $gid,my $rdev,my $size,
   my $atime,my $mtime,my $ctime,
   my $blksize,my $blocks)=stat($name);
  return $ino;
}

sub inode2name {
  my $inode=shift;
  opendir(DIR, '.');
  while (my $dname=readdir(DIR)) {
    (my $dev, my $ino,my $mode,my $nlink,
     my $uid,my $gid,my $rdev,my $size,
     my $atime,my $mtime,my $ctime,
     my $blksize,my $blocks)=stat($dname);
    if ($ino==$inode) {
      return $dname;
    }
  }
  return '';
}
```

```
sub pwd {
  my $inode=shift;
  (my $dev, my $ino,my $mode,my $nlink,
   my $uid,my $gid,my $rdev,my $size,
   my $atime,my $mtime,my $ctime,
   my $blksize,my $blocks)=stat('..');
  if ($ino!=$inode) {
    chdir('..');
    my $name=inode2name($inode);
    pwd($ino);
    print "/$name";
  }
}

pwd(name2inode('.'));
print "\n";
```

Unfortunately, this pwd code does not work properly with multiple disks or multiple partitions on the same disk. Each partition has it own file system tree. One file system is designated as the *root file system*. The root of its tree is the root of the entire tree. Each of the remaining file systems is attached to some subdirectory of the root file system. This is called *mounting*, and the subdirectory is called a *mount point*. Look carefully at the following directories.



The scenario above suggests two things:

- when '.' and '..' have the same inode, this does not necessarily mean that we have reached the root. Therefore, the recursion in the pwd function will stop short.
- when a file system is mounted, the inode of its root may be different than the entry corresponding to it in the parent directory. Therefore, the search in the inode2name function will fail.

(a) Fix the first problem by using chdir("..") and **stat**ing '.' instead of **stat**ing '..'

(b) Fix the second problem by using the st_dev field of the stat structure. So in some cases, instead of looking for a matching inode, you will be looking for a matching device.

Problem 2: Writing write

(a) Enhance the simplified version of the write command presented in the book on page 143 (does not send a greeting and requires the file name of the terminal). Your enhanced version must:

- accept the user name as argument
- send a greeting containing: the name of the user sending the message
- handle the cases when the user is not logged on, when the user is logged on at multiple terminals, and when the user is logged on but does not allow to me messaged

(b) Implement a command called mesg to flip the existing permissions on accepting messages at the terminals where you are logged on.

Hint: both parts can be done by making use of information in the utmp file. Alternatively, using the uid, one can list files in /dev/pts/ and identify the terminals corresponding to a given user.

Problem 3: Links for locks
Consider the following C++ function:

```
int lock(const char * name) {
  char * namedotlck = new char[strlen(name)+4];
  strcpy(namedotlck, name);
  strcpy(namedotlck+strlen(name), ‘‘.lck’’);
  int status=link(name, namedotlck);
  delete[] namedotlck;
  return status+1;
}
```

(a) Explain how you can use the above locking mechanism to make sure that if two processes access the same file, only one will succeed. What is it about the *link* system call that makes it a useful way to lock files?

(b) Implement a similar function to unlock a file.

(c) Write a program that, given a filename, repeatedly:

- writes the word “hello” to the end of the file
- writes the word “bye” to the end of the file

Run this program from two different terminals for some time, then stop them using Ctrl-C. Verify that sometimes the words are interleaved, e.g. hello hello bye bye.

(d) Use the locking and unlocking mechanism to avoid this interleaving.

(e) This problem illustrates how to lock an existing file, but how can a program use *link* to prevent two processes from both *creat*ing the same file?

PS: *link* and other file system functions are also available in Perl.