

# Homework 5

*Due: 13 Apr 2009*

## Problem 5.1

In this problem, you'll implement a program that largely mimics `id`. It takes one argument, a numeric user id. If the uid is not specified, it prints information about the current user.

Command to read up on: `id`

System calls to read up on: `getuid getpwuid getgrgid`

## Problem 5.2

In this problem, you'll implement a program that partially mimics `date`. With no arguments, it prints the current time and date in exactly the format used by `date`. If given one argument it is interpreted as a number of seconds since the epoch,<sup>1</sup> which is further interpreted with respect to our local time zone and printed in the same format as before. Note that, like `date`, you should be printing the time zone, so you won't be able to use `ctime` here.

Command to read up on: `date`

System calls to read up on: `time localtime strftime`

## Problem 5.3

Continue with your implementation of `ls`: implement the `-l` option (which should still operate with the `-a` and `-p` options, and either with or without a directory argument).

If you have not yet completed the previous two problems, your program can resort to the unsatisfying but perfectly acceptable strategy of displaying user, group, and modification time as raw numbers. If you have completed the previous problems, though, it should be fairly easy to have your program mimic the real "`ls`" pretty exactly.

On `timespec`: I can't find a man page that actually documents `struct timespec`, but it is defined as

---

<sup>1</sup>"The epoch", here, means midnight GMT on 1 January 1970. Google for more info.

```
struct timespec {
    time_t  tv_sec;   /* seconds since the epoch */
    long    tv_nsec; /* and additional nanoseconds */
};
```

or the equivalent. (So  $3\frac{1}{2}$  seconds after the epoch would have a `tv_sec` of 3 and a `tv_nsec` of 500,000,000.)

### Problem 5.4

In the course directory is a file named `hwk5_mystery.c`. Copy it into your own directory and modify it as much as you like without creating or removing any variables, changing the lengths of arrays, or changing the definition of the struct. Your goal is to determine precisely how the C compiler and Mac OS cause the memory to be laid out at the point marked `/* HERE */`. Specifically, what you should do is create a loop that modifies `ch` and prints the values it points to, walking right off the end of the array and into other parts of the stack, possibly adding other print statements to help you make correlations.

For this problem, you should hand in three things: first, on paper, a diagram of memory that accounts for every single byte and each of the nameable locations (including struct fields and array indices). Second, electronically, your modified `.c` file that enabled you to make that diagram. And third, either on paper or electronically, an explanation (possibly brief) of how you used the output from your modified program to draw the diagram, and of any spots that puzzled you or that you weren't able to explain.

Hint: you'll want to set variables to many different distinct values....

### Problem 5.5

(Based on a problem from the book.) Some operating systems provide a system call `rename` to give a file a new name. Describe some differences between using this call to rename a file and just copying the file contents into a new file with the desired name, followed by deleting the old file. (There are at least two major ones.) Are there tradeoffs, or is one technique always better than the other? Does the answer change if the underlying filesystem is contiguous, linked list, FAT, or inode-based? Why or why not?