# ECS150
## Operating Systems
Winter Quarter 2004

Lab 3
I/O Drivers, Memory Management, File System

**Due Tuesday, March 16, 12 Noon**

These problems ask you to make very small modifications to Minix. You will be modifying the kernel (actually I/O drivers), the memory management server, and the file system server, recompiling and assembling them using `MAKE` to link the various executable files, and then produce a new Minix boot disk.

Each of these programs requires only trivial modifications to Minix. The key is locating the current source code that should be modified; there are many acceptable solutions.

To receive credit for you solutions, you will follow the procedure Sophie will outline in a posting. Interactive grading will occur on T,W, and Th.

1. Modify the terminal driver to provide a primitive editing capability, much as appeared in the early line editors - before your time. When `control-p-n` is struck followed by a character `c`, the $n$-th character ($1 \leq n \leq 9$) back from the last character in the tty buffer is replaced by `c`; the rest of the buffer is unchanged. (What I mean by `control-p-n` is as follows: While the `control` key is being struck, `p` is struck and released followed by some single digit being struck and released.) You will have to decide on an action if $n$ is larger than the number of characters in the buffer; I suggest that no change takes place to the buffer. The contents of the edited line are displayed; you can do this by erasing from the display the previous line and replacing it with the new line. You will not need to run a program (other than Minix, of course) to demonstrate the new terminal driver.

2. Modify the `Memory Manager` so that a zombie's memory is released as soon as it exits and enters the zombie state rather than having the release wait until the parent performs a wait. To test out your new `Memory Manager`, you will have to run a program and use a hot key to display the memory maps. The program can be very simple, as all it has to do is create a child and ensure that the child exits before the parent performs a wait. You will use the hot key to display the memory map before and after the child exits, so make sure your child stays around long enough.

3. Modify the `FS` to provide a system call that appends two files, designated by their file descriptors `fd1` and `fd2`, where the result of the call is to replace the file designated by `fd2` so that the file designated by `fd1` appears at the end of the former. The file designated by `fd1` is to be unchanged. To avoid the need to modify the c-compiler, here is an easy way to implement the "append" system call. Use `lseek` as follows: Have the first argument to `lseek` be $-1$ (to indicate that `lseek` is being called in a non-standard way, have the second argument be `fd1`, and the third be `fd2`. Be sure to include error checking in your implementation, for example to check against unallocated (i.e., un-opened) file descriptors being arguments. Also, you should decide what you wish to do with the read-write pointers for the files involved in the

call; I suggest that after the call, the read-write pointers should point to the beginning of the file. You will have to run a simple program to test your new `FS`, for example open two files passed as arguments to your program, call `lseek` so it performs an append, and return success or failure (e.g., if the files do not exist or the process does not have the appropriate permissions).