

Lab 2

2 Apr 2009

In this lab, you will learn to program using calls to the operating system (as opposed to simply functions in the library). The task of the day will be a fairly straightforward file-copying tool, file management being one of the key jobs of the OS. Your program will behave much like the Unix command `cp`. Specifically, your program should take two filenames from the command line and copy the contents of the first file into the second.

The four main system calls you will be using are `read`, `write`, `open`, and `close`.

1 System calls and man pages

We've seen man pages before, and the man pages for system calls aren't terribly different from those for functions. There are a few points worth highlighting as you call up and read the man pages for the four system calls you'll use:

- System calls are in section 2 of the manual (while functions like `printf` were mostly in section 3). For words that head more than one page in the manual, you can use `man -a` as described before, but you can also skip straight to the one you want if you know what section it's in: `man 2 read` will take you to the `read` OS call (which is what you want) and skip over the shell command and the Tcl builtin.
- More than one system call may be documented on the same page; learn to skip the material you're not currently interested in.
- Notice the section `RETURN VALUES`. With system calls the usual return value is often some incidental value like the number of bytes transferred—or a special value (like `-1`) on failure. This return value is often *the only indication you have that an error occurred*. So you should get in the habit of checking it.
- Further note the section `ERRORS`, but don't dwell on this yet (we'll come back to it).

2 Basic file copying

When all is right with everything and goes according to plan, your basic file copy operation opens two files, reads from one and writes to the other, and then closes both files. For your first pass (as you're reading the manpages and figuring out the system), just mostly assume that the user is providing valid input and everything will work—that is, don't go out of your way to check for errors. You might need to read the sidebar below before parts of the open manpage make sense.

Once you think you have a handle on the normal operation of the four calls and have written the basic program, try it (but not with any files you wouldn't want to lose—an error here could erase the input file!). If it works, great. If not, move on anyway because the section on error detection and reporting will also be helpful for debugging.

Sidebar: Bitwise arithmetic and bit fields

The basic idea behind bitwise arithmetic is this: if you line up two integers and write them in binary, each bit of one is paired with a bit from another, and you can do the standard boolean operations (like AND or XOR) on them, yielding a result integer of the same length. For instance, if you bitwise AND the numbers 106 (hex 0x6A) and 78 (hex 0x4E), you get 74 (hex 0x4A):

```
01101010 (= 0x6A or 106)
01001110 (= 0x4E or 78)
=====
01001010 (= 0x4A or 74)
```

In C, there are four bitwise operators:

```
& bitwise AND
| bitwise OR
^ bitwise XOR
~ bitwise NOT
```

The basic idea behind a bit field or flag field is that sometimes you want to specify a whole bunch of mostly independent boolean options all at once; rather than pass a bundle of boolean variables, each named option is assigned a power of two—corresponding to a single bit—and then the bundle is passed as a single int value where the 'true' or 'yes' bits are set to 1.

For instance: let's say we're building a webpage and want to separately specify whether it contains audio, whether it contains video, and whether it contains interactive Flash. We could do something like

```
int HAS_AUDIO = 1; // = 001
int HAS_VIDEO = 2; // = 010
int HAS_FLASH = 4; // = 100
```

If a page has both audio and Flash (but no plain video) we say

```
int bits = (HAS_AUDIO | HAS_FLASH)
```

taking the bitwise OR of 001 and 100 to get 101. On the other end, someone with a bit field can ask whether the video bit is on by saying

```
bool video = (bits & HAS_VIDEO)
```

because the *only* bit that's on in `HAS_VIDEO` is the second one, so regardless of the audio and Flash values, if there is no video the result will be zero, or false. (If there is video, the result will be 2, but remember that this counts as true in C.)

3 Error detection and reporting

In general, when dealing with operating system calls, there are lots of things that can go wrong that you have no direct control over. You should program in a very defensive way by checking all the return values from these calls and handling errors gracefully. Even outside of the system calls, there are problems, best classified as 'user error', that you should check for before proceeding too far—for instance, what happens if you try to access a command line argument that hasn't been provided? Just as in Java, it's always better to intercept problems like that while you're still in control, and can handle it gracefully.

What does it mean to handle an error gracefully? Depends on the error, but it usually involves at least two things: reporting the error in terms the user can understand (more or less), and then either letting the user fix it or terminating the program before anything else can go wrong.

To report an error, you'll want to send a message to standard error, which you might recall is a lot like standard output (in that it usually goes to the screen) except that it doesn't normally get redirected even when standard output is. The easiest way to send your own message to standard error is with `fprintf`, much like `printf` except with an extra parameter at the front:

```
fprintf (stderr, "a < b, can't proceed: a=%d b=%d\n", a, b);
```

The other main way to send a message to standard error is if the problem is the direct result of a failed system call. Remember the **ERRORS** section of the manpage? Usually, if a system call fails, it will return `-1` (less commonly, `0`), and set a global variable deep in the system (called `errno`) to specify what went wrong. You can access it directly and compare it to the named labels (like `EACCES` and so forth), but much easier is to use the functions `perror` and/or `strerror` to do it for you. Note: these will only work if called right after a failed system call; at other times, the value of `errno` may be stale and unreliable.

Finally, if there is no meaningful way to proceed, the next thing you should do is terminate the program. If you are in the `main` method, you can simply `return` (and since you're in a failure condition, you should return something other than zero). However, you can immediately terminate the program from anywhere in your program with a call to `exit`:

```
exit (EXIT_FAILURE);
```

4 Wrapping up

If you find yourself with some extra time, add code that observes the `-i` and `-n` parameters of `cp`. Either way, though, submit whatever you do finish via the handin script:

```
handin cs226 lab2 whatever.c
```