Lab 4

Out: Friday, February 25th, 2005

What you'll learn.

In this lab, you'll learn to use function pointers in a variety of applications. You will also be using some basic image processing techniques to alter images in a generic filter function. By the end of this lab, you should also have a good grasp on forking a process and executing a program in the child process, and you should know how to send signals to child processes. Finally, you will learn how to handle (ie, receive) signals.

How you'll do it.

To perform the simple image processing tasks that we ask, namely invert and greyscale, you'll be writing a single generic function that takes as input a pointer to an image struct and a function pointer to an "operation" function. Using a generic function like this to perform these operations will save coding for you, and it will also allow the function to be used (hopefully) unaltered in later labs.

You will be using a lot of your code from previous labs. First, you will open an image file. You will then add code to `fork()` a process and `exec(..) xv`, which will open the image to view. You will then run a filter on the image, write it to the file, and call `kill(..)` to send a signal to `xv` to reload the image. In this way, you will use `xv` to display the image before and after the filter. Last, you will alter your code to rerun the filter and redisplay the image each time you kill -USR1 your process by catching SIGUSR1 using the `signal(..)` function.

1 A Generic Image Processing Function

A generic image processing function should take as input an image_t pointer and a pointer to an operation function.

```
typedef void filter_op( color_t * );

void filter( image_t * img , filter_op op );
```

This generic image processing function should loop through all the pixels in the image. At each pixel, it should call the operation function on the current pixel color. Thus the operation function's argument should be a pointer to a color_t.

Task: Write a `filter` function whose arguments are a pointer to an image struct and a pointer to an operation function. It should iterate through the pixels in the image and call the operation function on each pixel.

2 Operation Functions

Two of the easiest image processing operations that can now be written are invert and greyscale. The invert operation does pretty much what its name suggests – a value of 255 in the original image should be 0 in the filtered image. A value of 253 in the original image should be 2 in the filtered image. You perform this operation on the red, green, and blue channels.

The greyscale function takes a `color_t*` and averages the red, green, and blue values. That average should be stored in each of the three channels – the red, the green, and the blue.

```
void invert( color_t * c );
void greyscale( color_t * c );
```

Note that when you are performing operations like averaging the three colors, you must be careful when choosing which data type to use to store temporary information. It is possible to "overflow" certain data types. For example, trying to store "255 + 1 + 1" to an unsigned char variable is not going to store the number "257," since an unsigned char is only designed to hold values up to 255.

Task: Implement an invert function and a greyscale function. Both functions should take a pointer to a color_t as input.

3 Testing

You must prove to the TAs that both your invert and your greyscale filter work as described above (using function pointers).

Task: Modify old code or write a new main.c file that will load an image and perform one of the above operations on the image, and then save the resulting image to a file. Be prepared to prove to a TA that both functions work.

4 Child processes

In this section, you will learn how to create a child process and wait on it.

fork(2)

Say you're writing a shell. Shells do one thing well; they start running other programs. Basically, this is done in two steps: splitting your process into two and then, in one of the copies, changing which program is running. To create a new process, you call `fork()`. This is the part that gets a bit confusing. `fork()` returns the child's PID to the parent process; it returns 0 in the child process; `fork()` will return -1 (to the would-be parent) if the fork doesn't work.

```
if((pid = fork()) < 0){
     //some sort of error occurred
}else{
     if(pid == 0){
          //child goes on from here
     }else{
          //parent goes on from here
     }
}
```

exec(3)

At this point, you've got two processes running (almost) the same program. Now you'd like to change which program is running in one of the processes. This should be done in the child process. In the child process, you will want to call `exec(..)`. Here is an example of a call that will start running `xv` with no arguments:

```
execl("/usr/X11R6/bin/xv","xv",0);
```

There are a lot of variations of exec. Read the manpage (man 3 exec) for more information about which version of exec you may wish to use and how to use it. Note the files to `#include`. `exec(..)` and its variants will not return unless there was an error. That means this should be the last piece of code in the block labelled "child goes on from here" except for error-catching code.

wait(2)

You don't want your parent process to exit until `xv` closes. To ensure that, you should call `wait(..)` in the parent process on the child process PID. `wait(..)` will not return until `xv` closes.

xv(1)

`xv` is a simple program that can display images. The most basic way of using it is:

```
xv <filename>
```

There are some convenient options to `xv`. Unfortunately, as you may find with many free software, the man page is kind of empty. `xv -h` will display a bit of information. The option we find most useful for this assignment is `-expand` <`int`>. This will stretch the image to make it easier to see, which is good if you are working with small images. Play with the integer value to work for you. 10-20 are reasonable values for really tiny images.

> **Task:** Use your image filtering code from the previous tasks. You will still take the filename from the argument list for main(), but for the moment, you don't even need to read the file. Write a small function in ppm.c that will fork and exec `xv` to display an image from a file. The function should take as its sole argument the name of the file to have `xv` open, and it should return the PID of the child process or -1 on failure. In your main() function, you should have the parent process `wait(..)` on the child process before your program ends. When the user closes `xv`, your program should close. Make sure to add the function prototype to ppm_header.h.

5 Sending signals

In this section, you will learn how to send signals to child processes.

kill(2)

The way to send a signal is to call `kill(..)`. `kill(..)` takes the PID of the process to signal and the signal number to send. For this part of the lab, you will be sending signal SIGQUIT to `xv`. This tells `xv` to reload the image it is displaying. (That's a very xv-specific thing.) You must use the PID acquired from the call to `fork()` that you made in the previous task.

> **Task:** Modify your program such that it now reads an original ppm from a file and has `xv` display the file. Keep the original image in memory. After the user types enter into the term, your program should run a filter on the image, preferably non-idempotent,[1] write it out to the same file, and signal `xv` to redraw the image, which should now be the filtered image. You should then wait on the `xv` process. Don't worry about the case when the user closes `xv` before he/she types enter into the term.

[1] In math, idempotency refers to functions in which repeated applications have the same result as one. The same situation applies here: Use a function in which the image changes every time you apply the filter. Invert is a good example.

6 Handling signals

In this section, you will learn how to handle signals received from other processes.

When you run a program (like the one you are writing in this lab) from a shell, and your program is in the foreground of the shell, some signals you send to the shell get redirected to the program running in the shell. An important example of this when you hit ctrl-c in a terminal. The shell running inside the terminal will send a SIGINT to your program. By default, programs will quit when you do this. However, maybe your program would like the opportunity to clean things up before it quits. In that case, you will need to write a signal handler for that signal. In fact, in the case of this task, your program won't even quit.

signal(2)

A signal handler is nothing more than a function. In order to install the handler, you use the function `signal(..)`. `signal(..)` takes in which signal the function will handle and the function name. There is a limitation on the function. As found in man signal(2), signal handlers are defined as:

```
typedef void (*sighandler_t)(int);
```

That basically means that the function is void (where would it return values to, anyhow?) and only takes one argument, an int. The int that the function takes in will be the integer value associated with the specific signal that occured.

kill(1) and the shell

kill isn't just a C call. There is a shell equivalent. For this part of the lab, you will send signals to your program using kill from a shell. You will be sending signal SIGUSR1. Do it like so:

```
kill -USR1 <pid>
```

where <pid> is the pid of your program. You can get the pid of your process with getpid(2).

> **Task:** Modify your program by adding a signal handler that applies your filter to the image in memory, writes it back to the file, and sends a signal to `xv` to redraw the image. Install this handler for signal SIGUSR1. Print out your process ID (PID). Now, each time you send signal USR1 to that pid using the unix command `kill`, the filter should be applied to the image and `xv` should show the results. When the user types <enter> in the term your program is running in, everything should close including the `xv` process. Do this by sending a SIGTERM signal to the child process. Remember to wait on the PID of `xv` right after! Again, don't worry about the case when the user closes the `xv` window.