CS 240 Summer 2017

Lab 7: Putting It All Together (130 pts)

Due: 08/01/2017 (Tue), 11:59 PM

Objective

The objective of this lab is to "put it all together" and make use of key C programming techniques. This includes run-time methods that allow running programs to exercise control over what they do and how they do it.

Reading

Read chapter 6 from K&R (textbook).

Lab 7 Code Base

The C code base for lab7 is available as a tarball

/u/data/u3/park/pub/cs240/lab7.tar

on the lab machines.

Problems [130 pts]

Problem 1 (130 pts)

This is an extension of Problem 4 in lab3 where the vectors are allowed to be complex, that is, the components of the vectors are complex numbers. The rule for adding two complex two complex numbers x + i y and a + i b is

$$x + a + i (y + b)$$

where i represents the imaginary number sqrt(-1). The rule for multiplying two complex numbers x + i y and a + i b is

$$(x + i y) (a + i b) = x a - y b + i (x b + y a)$$

More simply, when manipulating complex number we may treat the first complex number as a pair of numbers (x,y) and the second complex number as (a,b). The imaginary number i is not needed when processing complex numbers. The rule for adding two pairs of numbers (x,y) and (a,b), when viewed as representing complex numbers, is

$$(x,y) + (a,b) = (x + a, y + b)$$

By our definition, (x + a, y + b) represents the complex number x + a + i (y + b). The rule for multiplying two pairs of numbers (x,y) and (a,b), when viewed as representing complex numbers, is defined as

$$(x,y)\ (a,b) = (x\ a - y\ b, x\ b + y\ a)$$

Treating a complex number as a pair of numbers (i.e., tuple) and defining addition and multiplication of tuples (when viewed as representing complex numbers) as above simplifies their manipulation in computer programs.

Now, when the input vector of size (i.e., dimension) N is allowed to be complex, it means that we have N pairs of numbers. For example, if N = 3 then the format of the input typed on stdin is

```
3
11 2 50 77 12 4
7 26 831 65 31 41
```

which means the first vector has components (11,2), (50,77), (12,4) (viewed as complex numbers), and the second vector has components (7,26), (831,65), (31,41). These kind of calculations often occur in various apps including computer graphics (e.g., games) and programs that control physical systems (e.g., autonomous vehicles).

You will need to modify the previous functions for reading and calculating dot product to deal with complex numbers where each component of a vector is a tuple (i.e., itself a 1-D vector of 2 components). Rename the functions read_vectors_complex() and calc_dotprod_complex(). You do not need to calculate the magnitude (i.e., norm) of the input vectors which implies that calc_dotprod_complex() has two fewer arguments than calc_dotmag(). To allow vector components to be tuples, make use of C's struct and typedef where, first, define a composite data structure representing a complex number as

```
typedef struct complex_num {
  float x;
  float y;
} complex_num_t;
```

Then define a complex vector as a 1-D array a[] of type complex_num_t. Unlike Problem 4 of lab3, use malloc() to allocate just enough memory at run-time for the vectors. The overall structure and logic of your app remains the same.

Another change is the UI (user interface) of your app which supports command-line arguments. Suppose your app's binary is called dotprodcomplex. Instead of reading vector size N and two vectors from stdin, the relevant information will be given as command-line arguments to dotprodcomplex. The format is:

```
% dotprodcomplex 3 vector1.dat vector2.dat
```

where 3 is the vector dimension and vector1.dat is an ASCII file containing

```
11 2 50 77 12 4
```

which represents the first vector in the preceding example. Similarly for vector2.dat. Output is written to stdout. Implement and test your code, and submit your work in v10/.

Bonus Problem (80 pts)

This is an extension of Problem 2, Lab 6, that adds the capability of your shell to support "background processes". For example, if the user sue101 enters

```
sue101: ls -l -a &
```

then the binary ls should be run "in the background" meaning that the shell immediately returns a prompt and waits for the next user input while the background process is executing the code of ls. Here "sue101:" is a new prompt that shows the username/loginID of whoever is running the shell (instead of the PID in our earlier shells discussed in class). Use execvp() to execute the requested binary. When you run ls in your login shell on the lab machines, you will notice that the login shell will print to stdout a message when a background process terminates to notify the user of this event. We will do the same, albeit with our own message format. For example, the following

```
sue101: sleepbin &
sue101:
pid 8772 has terminated: exit 0
sue101:
```

shows an instance of user interaction with your shell, where sleepbin is a program that sleeps for 10 seconds (by calling sleep()) and exits with exit value 0 (the normal exit condition) by calling exit(0). Your shell runs sleepbin in the background and immediately returns the prompt. Suppose the user does not enter another command for 10 seconds at which time sleepbin terminates. Your shell alerts the user by printing a message to stdout that specifies background process with pid 8772 has terminated and its exit value is 0. A fresh prompt "sue101: " is output to facilitate a clean user interface.

To support background processes in the sense above (Linux background processes have additional properties that are outside our scope), we need to achieve the following: the shell (parent process) needs to find out when the background process (child process) terminates. In addition, the parent process needs to access the child's exit value. To do so, we will utilize the signal facility provided by Linux. As discussed in class, you will write a signal handler, call it, mon_chld() with function prototype

```
void mon_chld(int);
```

that is registered with the Linux kernel by calling signal() with first argument SIGCHLD and the address of mon_chld() (i.e., function pointer) as the second argument. Subsequently, whatever the shell process may be doing, Linux will call mon_chld() when a child process terminates. Your code of mon_chld() will call wait() which returns the pid of the child. To determine the exit value of the child process, you will call wait() with a pointer to an int variable as its argument which wait() uses to store the exit value of the child process (among other things). You access the exit value by using the macro WEXITSTATUS(). Test your background process enabled shell with various binaries including sleepbin (write your own version) that sleeps for 7 seconds before terminating by calling exit(1). Submit your work in v11/. When coding your shell, please keep track of the myriad system header files that need to be included.

Turn-in Instructions

Electronic turn-in instructions:

i) For code submissions, follow the directions specified in the problems.

ii) Use turnin to manage lab assignment submissions. Rename lab7 to mylab7. Go to the parent directory of your mylab7 directory and type the command

turnin -v -c cs240 -p lab7 mylab7

Please note the assignment submission policy specified in the course home page.

Back to the CS 240 web page