CS 240 Summer 2017

Lab 6: Concurrent Client/Server Programs and Dynamic Memory Allocation (290 pts)

Due: 07/26/2017 (Wed.), 11:59 PM

Objective

The objective of this lab is to further develop concurrent client/server C programming skills and make use of dynamic memory allocation in Linux processes.

Reading

Read chapters 7 and 8 from K&R (textbook).

Lab 6 Code Base

The C code base for lab6 is available as a tarball

/u/data/u3/park/pub/cs240/lab6.tar

on the lab machines.

Problems [290 pts]

Problem 1 (70 pts)

This is an extension of Problem 1, Lab 5. Instead of using

```
#define MAXSIZE 50
```

inside dotheader.h to predefine the maximum size of arrays that hold input and output vectors, use malloc() to allocate the precise amount of memory needed by the app process at run-time. That is, after reading the first number from the input file vecin.dat that specifies the size of the vectors, use malloc() inside read_vectors() to allocate contiguous memory for the 1-D arrays as discussed in class. Since read_vectors() is modified to accomplish dynamic memory allocation, consider carefully if its function prototype needs to be changed as well.

The legacy vector calculation functions remain backward compatible and need not be modified. Reuse their existing object files when creating an updated shared libary using Makefile. Also, use variable definitions such as

```
CC = gcc
OBJ = main.o read_vectors.o calc_dotmag.o calc_diff.o calc_power.o
```

and CFLAGS, among others, that you find useful to create a more transparent, modular and extensible Makefile. For example, when using a different C compiler on another development platform, only the definition of CC needs to be changed. The variables are referenced using $(CC) and $(OBJ) inside Makefile. Submit your work in v10/.

Problem 2 (120 pts)

This is an extension of Problem 3 in lab5 that allows esh to execute binaries with command-line arguments. That is, "ps -e -f" is now allowed. In lab6.pdf, explain how you are reading and parsing user input from stdin to prevent stack overflow/smashing. To handle command-line arguments, use the function execvp() in place of execlp(). execvp() takes the name of an executable binary as its first argument. Its second argument is an array of pointers to strings. For example, for "ps -e -f"

```
char *s[] = {"ps", "-e", "-f", NULL};
execvp("ps", s);
```

will overlay a process's binary with ps with options -e and -f. Note that the first argument to execvp() is the name of the binary to be loaded/executed but so is the first element of the array of string pointers, s[0]. The last element of s[] must be NULL. Of course, in your implementation of esh the name of a binary and its command-line arguments (if any) are provided as input by the user via stdin. Use string processing library functions to streamline parsing of user input. Make sure to include string.h when doing so. Make sure to check the return value of function calls, including fork() and execvp(), to create a bug free program. Be vigilant as the TAs will look for ways to break your program. Test your extended shell with binaries containing command-line options. This includes "mv" that must have two file names as command-line arguments, "host", and "time" whose argument is an executable. After running the executable, time prints to stdout how much CPU time (among other resources) the executable has consumed. Submit your work in v11/.

Problem 3 (100 pts)

Write a function CloneRunApp() of function prototype

```
int CloneRunApp(char *binary)
```

that combines the tasks carried out by fork() and execvp() when used to spawn a child process that executes a binary specified in the argument that may contain command-line arguments. CloneRunApp() takes as argument the name of an executable binary with command-line arguments, forks a child process, and executes the requested binary using execvp() in the child. When execvp() is successful, the child process executes the requested command. If not, the child process terminates by calling exit(1). The parent process, unlike in the concurrent server design of a shell, does not wait for the child using waitpid(). Instead, it returns immediately to the caller with the PID of the child process. For the moment, we will not be concerned that not calling waitpid() will turn the child process into a zombie when it terminates (zombies are real in the computing world) which is undesirable but not critical. If fork() fails, CloneRunApp() returns -1. For testing purposes, put the code of CloneRunApp() in a single file CloneRunApp.c. Check that the function works correctly. After verifying correctness, re-implement esh of Problem 2 so that it uses CloneRunApp() instead of calling fork() and execvp(). The CloneRunApp() function should make it easier for a C programmer to code esh (and other concurrent client/server apps) by hiding the underlying mechanisms involving fork() and execvp(). Submit your work in v12/.

Bonus Problem (20 pts)

An extension of Problem 3, add a delayed execution capability where a user, via an internal shell command to esh, can request that a command be executed t seconds from now. For example, if a user types

```
$ dd17 host pod2-2.cs.purdue.edu
```

then esh executes "host pod2-2.cs.purdue.edu" 17 seconds in the future. Delayed execution is accomplished by calling sleep() in the child process before execvp() is called. If delayed execution is requested of the shell, the parent should not call waitpid() but print a prompt and block on user input from stdin. Look up the man page of sleep() to determine how to use it. Submit your work in v13/.

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 lab6 to mylab6. Go to the parent directory of your mylab6 directory and type the command

```
turnin -v -c cs240 -p lab6 mylab6
```

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

Back to the CS 240 web page