Programming Assignment #1

CPS 210
Due Date: 2/10 at 11:59 PM

(Available in postscript.)

This is a warm-up assignment to exercise the Unix system call interface and your C programming skills. Since the Unix interface is reasonably standard, you may use any Unix systems in the department to write and debug your programs (but tell us which types of systems you used). You are welcome to use trusted ARI machines (Suns and Alphas) for this assignment. If you wish, you may do this project in groups of two.

You will turn in all of your code in a single (one-level) directory. Your directory will include a single makefile to build your programs using **make** or **gmake**. Just typing **make** with no arguments should build all of your code. Your makefile should also include a **clean** directive to remove all of your object files, and a **scratch** directive to remove all files (objects, executables, and archives) produced by the build. You will turn in your directory with all objects fully built. Please include a **Readme** file briefly describing your programs. Directions for turning in your code will be provided before the due date.

Each program will be a few screens of code. Your programs should not have any bugs. You get to decide what is a "bug" and what is a "feature", but explain any restrictions in the block comment at the top of each program (note: dumping core is always a bug). Your code should show good Unix programming style. For example, all system call errors should be detected and reported with *perror* in a format that explains which operation failed and why.

Program 1: A Shell Game

Write a **minishell** program that executes specified programs as child processes, connecting them with a pipeline. The command syntax is **minishell <prog1> <prog2> <prog3>**, where the "progs" are the executable file names of programs that read from standard input and write to standard output. Your shell will set up the first process to read from minishell's *stdin* (e.g., the terminal), and the last process to write its output to minishell's *stdout*. Any standard Unix program should run fine under your shell, even if it uses environment variables.

You will use the Unix mechanisms for process groups and job control to handle keyboard-generated signals properly for your child processes. If you run **minishell** with its *stdin* bound to the keyboard, then typing **ctrl-c** or **ctrl-z** should interrupt or stop all the child processes, but not the minishell. To do this, your minishell program will need to create a separate process group for its children, and make the process group the *foreground* process group for the terminal. **Note**: your minishell need not and should not declare signal handlers for **ctrl-c** or **ctrl-z**.

Your minishell should print out the process IDs and program names for its children as it creates them and initializes them, and it should print the process IDs and exit status values as the children exit. If child processes pause (e.g., due to **ctrl-z**), the minishell should print the process IDs of the stopped processes and wait for them to resume (you may need to use the **kill** command from a separate window to resume them with a SIGCONT signal to the process group).

To make your job easier, we have put some sample code for fork/wait, pipes, foregrounding, and other related Unix stuff in the directory **/usr/project/courses/cps210/sample_progs**. These programs are pretty solid, but they are not guaranteed to do the right thing on every system. In particular, the details of handling signals are different on Solaris and other SystemV-flavored Unix systems. We recommend that you use BSD-flavored systems, e.g., Digital Unix or FreeBSD(trusted machines do not yet exist for this assignment). The sample programs have been tested under Digital Unix.

Program 2: M-Pipes (more pipes)

In this assignment you will implement pipes utilizing the mmap facility. You should implement the following calls, m_pipe, m_read, m_write, and m_close. These calls should have semantics similar to their ordinary pipe equivalents. I.e. your calls should return error codes, and data should not be corrupt and should be delivered in order. The implementation should be simple; you only have to worry about setting up a pipe between two processes. Issues you may wish to consider: What sort of synchronization will you need? Do you have to worry about COW's (copy on writes)?

After you get that working, you should write a small test program, called **piper**, to evaluate your pipe implementation against the pipes you used in program 1. Then write about it. What performance differences did you notice and why? What are the key differences and similarities between your m_pipes and ordinary pipes?

Program 3: Heap Manager

Implement a heap manager consisting of your own versions of *malloc* and *free*, to replace the versions provided in the standard library. The virtual memory for your heap can be obtained via the *sbrk* system call. The heap manager should self-initialize the first time *malloc* is called.

Unlike the standard *malloc* and *free*, your heap manager may assume that the calling program will use a relatively small number of different heap block sizes, and that it will use each heap block size a large number of times. In other words, you may assume that the calling program will commonly use a few different block sizes, but that it has the liberty of allocating any size block if it so desires.

Include a test program called **heap** in a separate source file. **Heap** should treat each allocated heap block as an array of *long*, and read and write each heap block as necessary to test for corrupt data or bad pointers. **Heap** should take as arguments the heap block sizes to use for testing the heap.

Write a one-page evaluation of the choices you made in the design of your heap manager. Explain why and how your heap manager will perform better than an “ordinary” heap manager in the case where the calling program has the expected behavior. What tradeoffs did you make? How will your heap manager perform for an application that does not match the expected behavior? Consider both space and time overheads in your answers.