Episode 5: The UNIX Shell (Due: XX/XX/XX)

Introduction:

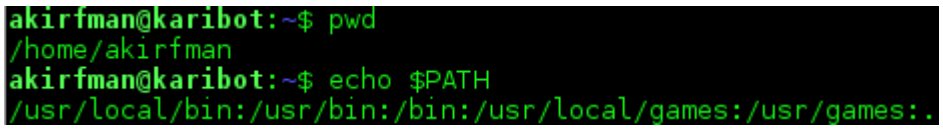
Most useful interaction with a UNIX system occurs through the shell. Using a series of easy to remember and simple commands, one can navigate the UNIX file system and issue commands to perform a wide variety of tasks. Even though it may appear simple, the shell encapsulates many significant components of the operating system.

Basic Shell Features:

Environment

The shell maintains many variables which allow the user to maintain settings and easily navigate the filesystem. Two of these that are particularly important are the current working directory and the PATH. As its name implies, the current working directory variable keeps track of the user's current directory. The PATH variable consists of string of colon separated pathnames. Whenever you type a name of a command, the kernel searches in the directories specified by the PATH variable starting with the leftmost directory first. If the executable is not found in any of the specified directories, then the shell returns with an error. One may modify the PATH at any time to add and remove directories to search for executables.

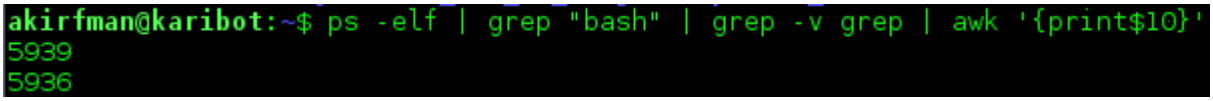
Figure #1: Current Directory & PATH



Pipelining

UNIX provides a variety of useful programs for you to use (grep, ls, echo, to name a few). Like instructions in C++, these programs tend to be quite effective at doing one specific thing (Such as grep searching text, ls printing directories, and echo printing text to the console). However, programmers/OS users would like to accomplish large tasks consisting of many individual operations. Doing such requires using results from previous steps in order to complete a larger problem. The UNIX shell supports this through the pipe operation (represented by the character |). A pipe inbetween two commands causes the standard output of one to be redirected into the standard input of another. An example of this is provided below, using the pipe operation to search for all processes with the name "bash".

Figure #2: Piping Between Commands



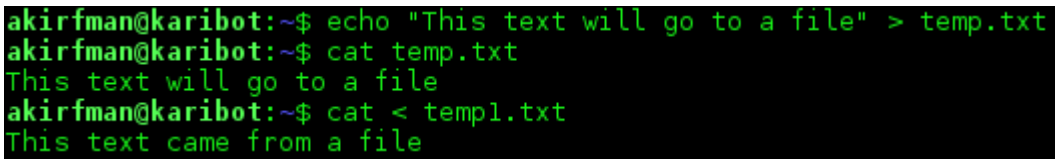
Input/Output Redirection

Many times, the output of a program is not intended for immediate human consumption (if at all). Even if someone isn't intending to look at the output of your program, it is still immensely helpful to have it print out status/logging messages during execution. If something goes wrong, those messages can be reviewed to help pinpoint bugs. Since it is impractical to have all messages from all system programs print out to a screen to be reviewed at a later date, sending that data to a file as it is printed is desired.

Other times, a program might require an extensive list of input commands. It would be an unnecessary waste of programmer time to have to sit and type them out individually. Instead, pre-written text in a file can be redirected to serve as the input of the program as if it were entered in the terminal window.

In short, the shell implements input redirection by redirecting the standard input of a program to an file opened for reading. Similarly, output redirection is implemented by changing the standard output (and sometimes also standard error) to point to a file opened for writing.

Figure #3: Input/Output Redirection



Assignment:

For this assignment, you are to design a simple shell which implements a subset of the functionality of the Bourne Again Shell (Bash). The requirements for your shell are as follows:

- Continually prompt for textual user input on a command line.
- Parse user input according the the provided grammar (see below)
- When a user enters a well formed command, execute it in the same way as a shell. You must use the commands fork and exec to accomplish this. You may NOT use the C++ system() command.
- Allow users to pipe the standard output from one command to the input of another an arbitrary number of times.
- Support input redirection from a file and output redirection to a file.
- Allow users to specify whether the process will run in the background or foreground using an '&'. (Commands to run in the foreground do not have an '&', and commands that run in the background do)
- (Bonus) Allow users to specify a custom prompt which supports printing the current directory, username, current date, and current time.

Figure #4: Simple Shell Grammar

valid_string = unix_command PIPE unix_command || unix_command REDIRECTION filename || unix_command AMP || unix_command || special_command

unix_command = command_name PIPE command_name ARGS

special_command = cd DIRECTORY || exit

command_name = any valid executable/interpreted file name

AMP = &

ARG = string

ARGS = ARG ARGS || ARG

DIRECTORY = absolute path || relative path

PIPE = |

REDIRECTION = < || >

Advanced Concepts:

File Pathnames

The UNIX filesystem is organized as a giant tree. Leaf nodes are files, and non-leaf nodes are directories which either contain files or other directories. The highest node in the tree is root, denoted by a forward slash (/). All other files/directories in the system are child nodes of the root node.

Filenames in a UNIX system are specified by a pathname (such as /bin/bash stating that the file bash is inside the directory bin which is inside of root). Pathnames are provided either as absolute pathnames, which are given relative to the root of the filesystem (Ex: /usr/bin/env), or as relative, which are specified relative to the current working directory (../../andrew/homework/CSCE_315).

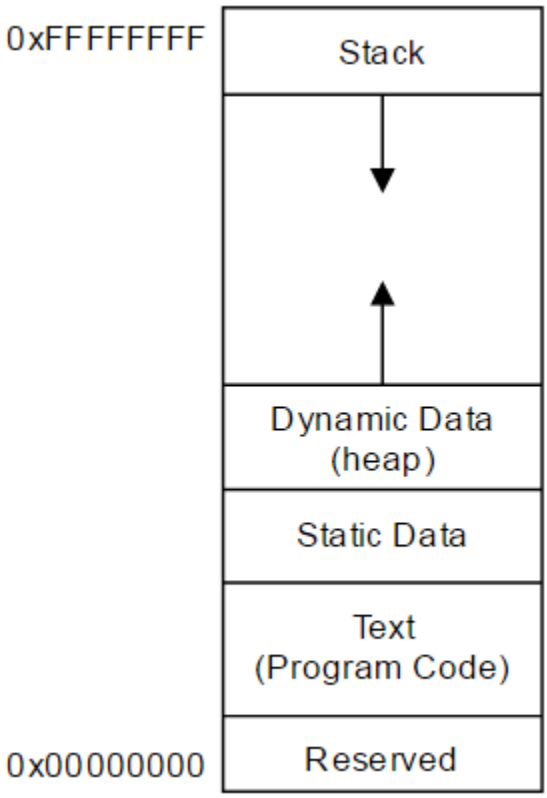
Side note: UNIX contains two special paths present in each directory. A single dot stands for the current directory, and two dots stands for the parent directory. To see these, type the command ls -l into a shell.

Process Address Space

The address space of a process is usually divided up into 4 regions, stack segment, dynamic data, static data, and text segment (See Figure #4).

All code pertaining to the execution of the program is contained in the text segment. Program data is stored in two sections depending on the type. Static data (global variables and constants) is defined before the execution of the program. As a result, this segment is of a fixed size which can be allocated on loading. Dynamic data contains all data allocated to the process through calls to malloc/new. Since this changes continually throughout the process's life cycle, this section of the address space grows towards higher addresses and shrinks back towards lower addresses. Finally, the stack contains data pertaining to function calls. As with the dynamic data segment, the stack must grow and shrink with the execution of a process. Therefore, the stack grows downwards from high addresses towards lower ones and shrinks back to high addresses.

Figure #4: Simple Address Space



fork();

When the operating system is initialized, two processes are created. The first, known as the swapper (process id 0), serves as the scheduler for jobs on the system. The second, known as init (process id 1), **More about init here!!!** Many times, it is necessary **Describe fork() here!**

```
algorithm fork
input:   none
output:  to parent proces, child PID number
         to child process. 0
{
    check for available kernel resources;
    get free proc table slot and unique PID number;
    check that user not running too many processes;
    mark child state as "being created";
    copy data from parent proc tabgle slot to new child slot;
```

```
    increment counts on current directory inode and changed root (if applicable);
    increment open file counts in file table;
    make copy of parent context (PCB, text, data, stack) in memory);
    push dummy system level context layer onto child system level context;
        dummy context contains data allowing child process to recognize
        itself, and start running from here when scheduled
    if(executing process is parent process)
    {
        change child state to "ready to run";
        return(child ID);
    }
    else
    {
        initialize PCB timing fields;
        return(0);
    }

}
```

The Six Exec Functions

After executing fork(), you are left with two identical processes. This isn't usually particularly useful unless you specifically need two exact copies of a process. Instead, UNIX provides a function, known as exec, which allows you to change a process's address space in order to run an entirely new program. Running an exec command deletes the existing text, data, and stack segments of the existing process and replaces them with with those of a new program. The algorithm for exec is presented below (Bach 218).

```
algorithm exec
input:    (1) file name
          (2) parameter list
          (3) environment variables list
output: none
{
    get file inode (algorithm namei);
    very file is executable and user has permission to execute;
    read file headers, check that it is a load module;
    copy exec parameters from old address space to system space;
    for(every region attached to process)
    {
        detach all old regions (algorithm detach);
    }
```

```
    for(every region specified in load module)
    {
        allocate new regions (algorithm allocreg);
        attach the regions (algorithm attachreg);
        load region into memory if appropriate (algorithm loadreg);
    }
    copy exec parameters into new user stack region;
    special processing for setuid programs, tracing;
    initialize user register save area for return to user mode;
    release inode of file (algorithm iput)
}
```

In UNIX systems, there is only one algorithm for exec, but the system call interface provides a total of six different variations of the exec function which differ only on how they handle input arguments. Short descriptions of each version along with short examples are provided below (Setvens 207-209).

Exec Functions & Examples

```
#include<unistd.h>

1. int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */ );

    Example:
    execl("/bin/bash", "bash", "-c", "echo \"Hello World!\"", NULL);

2. int execv(const char *pathname, char *const argv[]);

    Example:
    argv = {"/bin/bash", "-c", "echo \"Hello World\""}
    execv("/bin/bash", &argv);


The following exec commands take an environment array (envp) in addition
to the other arguments. This can be used to define a custom environment
for the process. Otherwise, uses parent's environment.

3. int execle(const char *pathname, const char *arg0, ... /* (char *) 0 */ ,
    char *const envp[]);

4. int execve(const char *pathname, char *const argv[], char *const envp[]);
```

Exec commands ending in a p use the PATH environ variable in order to find the executable file. An error is thrown if the file is not in a directory specified by PATH.

5. **int** execlp(**const char**∗ filename, **const char** ∗arg0, ... /* (char *) 0 */);

6. **int** execvp(**const char** ∗filename, **char** ∗**const** argv[]);

All return -1 on error and do not return when successful

UNIX man Pages:

One incredibly useful feature of UNIX operating systems that many new developers do not know about is the built in manual system. Using the command 'man', you can access information about most aspects of the operating system from general commands all the way to system call APIs.

The structure of man pages in UNIX are organized into sections by number follows (Wikimedia Foundation):

1. General Commands
2. System Calls
3. Library Functions (Specifically, the C standard library)
4. Special Files
5. File Formats
6. Games
7. Miscellanea
8. System Administration

You may (or may not, that's fine too) find the following manual pages useful when creating this assignment. Each one of these lines can be executed as a valid shell command to open a particular manual page. Note, the number indicates the manual section that that function resides.

- man 3 exec
- man 2 fork
- man 2 chdir
- man 2 pipe
- man 2 dup

Bibliography:

[1] Bach, Maurice J. *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice-Hall, 1986. Print.

[2] Stevens, W. Richard. *Advanced Programming in the UNIX Environment.* Reading, MA: Addison-Wesley Pub., 1992. Print.

[3] "Man Page." Wikipedia. Wikimedia Foundation, n.d. Web. 18 May 2016.