The Shell, System Calls, Processes, and Basic Inter-Process Communication

Michael Jantz

Dr. Prasad Kulkarni

Shell Programs

- A shell program provides an interface to the services of the operating system.
- It interprets user commands and uses the system calls provided by the operating system to execute commands.
- As programmers sought to increase efficiency and convenience of shell programs, shells with fairly sophisticated command languages developed.

Example Shell Commands

- Some common bash operators:
  - *foo* < in.txt – redirect standard input of program *foo* to in.txt.
  - *foo* > out.txt – redirect standard output of program *foo* to out.txt.
  - *foo* >> out.txt – redirect standard output of program *foo* to be appended to the file out.txt.
  - *foo* | *bar* – redirect standard output of program *foo* to be the standard input to program *bar.*
- Bash is also a scripting programming language (complete with variables and if and while statements) that can be used to script the OS.

Utility Programs

- Any Unix distribution comes with several utility programs for interacting with the OS.
  - grep – search for strings in a file
  - find – find a particular file
  - du – Determine the disk usage of files and directories
  - ls – List files and their permissions
- Many, many more. Proficiency with these basic tools will make you a much more effective developer on your platform.
- Unix provides a manual (accessible from the shell) that documents the use and syntax of each core utility. e.g:
  - man grep

finder.sh

```
find $1 -name '*'.[ch] | xargs grep -c $2 | sort -t : +1.0 -2.0 --numeric --reverse | head --lines=$3
```

- find $1 -name '*'.[ch] – Find files with .c and .h extensions under the directory given by the first argument.
- xargs grep -c $2 – Search the set of files on standard input for the string given by the second argument. -c says that instead of printing out each usage in each file, give me the number of times $2 is used in each file.
- sort – Sort standard input and print the sorted order to standard output. -t : +1.0 -2.0 says sort using the second column on each line (delimited by the ':' character) as a key. --numeric says to sort numerically (as opposed to alphabetically). --reverse says sort in reverse order.
- head – print only the first *n* lines of standard input. --lines=$3 lets us set the number of lines with the third argument.

How the Shell Works

- When the user types this command at a shell, the shell parses the input, and issues system calls to create the processes and set up the pipes between these processes.
- In this lab, we will implement what the shell would typically perform when given a command like this.
  - Although to save time, our implementation will only work for pipelines of length 4 as opposed to arbitrarily long pipelines (as a shell would handle).

Getting Started

- The first thing to notice after untarring the tar file is the Makefile:
  - Notice the variables DIR, STR, and NUM_FILES and the command under the 'find' target.
  - Test the command. In this lab's directory, do:

```
bash> make find
```

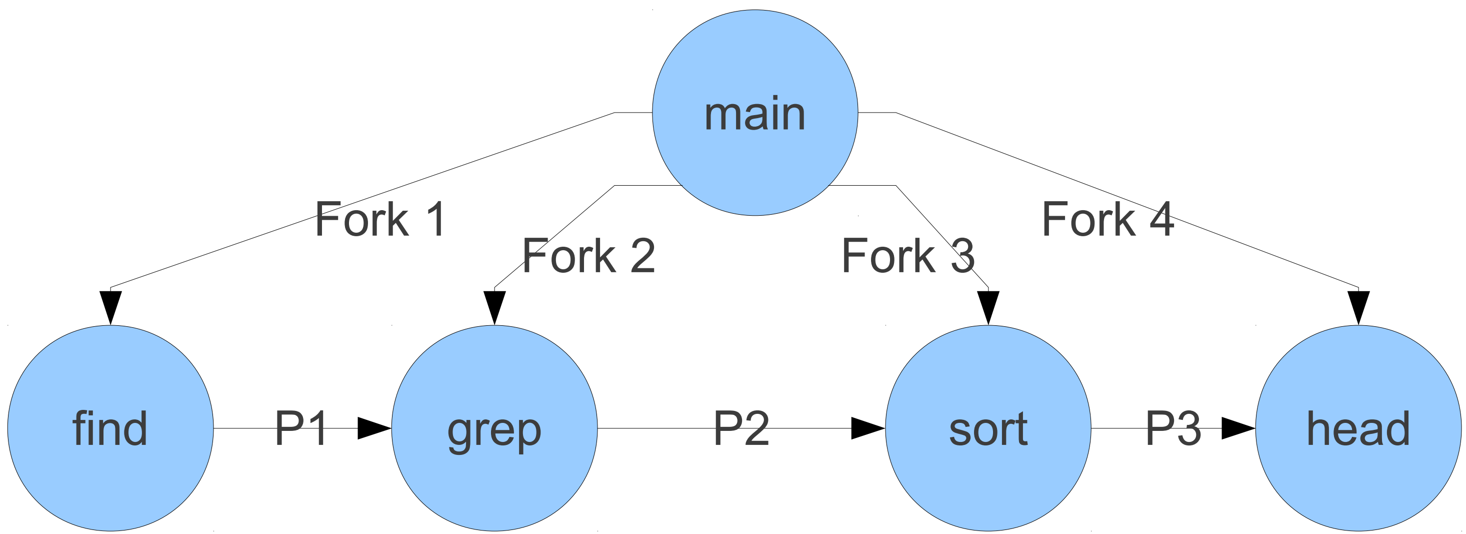
- Should see the output as described two slides back.
- The goal of this lab is to write a program – finder.c – that produces the same output as this command.

finder.c

- As it is given, this program is a skeleton for a four stage pipeline.
- All it currently does is start a process, which forks off four children (which do nothing), waits for them to finish, and exits.

finder.c (cont.)

- We want a program that forks off four children, sets up pipes between these children, and executes the appropriate command with each child:



System Calls

- In order to accomplish this, you will need to make use of a few different system calls.
- The starter code uses fork() and waitpid() to create the general framework under which you will implement the desired functionality.
  - You can look into the small program fork.c to understand the difference between the return values of fork() for parent and child
  - If you do not understand how these work, please ask me.
- You will extend this program to create the desired functionality using:
  - pipe(), dup2(), execl(), and close()

File Descriptors

- Before getting into how these system calls work, you must first understand file descriptors:
  - A file descriptor is a per-process, unique, non-negative integer used to identify an open file for the purpose of file access.
  - System calls (such as open, close, read, and write) use file descriptors to identify a particular file or pipe.
- **IMPORTANT:**
  - Each process has its own file descriptor table that maps each open file descriptor to a file object maintained by the operating system.
  - The file descriptor table is located in the process' PCB and is inherited by children of the process.

pipe()

- int pipe(int pipefd[2])
  - Creates a unidirectional data channel that can be used for interprocess communication.
  - pipefd[0] is the read end of the pipe
  - pipefd[1] is the write end of the pipe
- See man pipe for example usage.
- Things to think about:
  - How many calls to pipe should you make to construct a three process pipeline?
  - Which process(es) should create the pipe(s)? If you find yourself unsure, ask yourself which process(es) would create the pipe(s) if you were designing a shell that could handle pipelines of arbitrary length.

dup2()

- int dup2(int oldfd, int newfd)
  - On return, the newfd provided will be a copy of oldfd (i.e. newfd will refer to the same file (or pipe) as oldfd).
- Things to think about:
  - Each utility program we want to use in the pipeline reads from stdin and writes to stdout.
  - Why have I showed you dup2() and not the closely related dup()? See man dup for an explanation of dup.

Constructing the Pipeline

- Go ahead and try to setup the pipeline using the pipe() and dup2() system calls.
- Remember to close() unused file descriptors for each process.
- You may want to experiment with pipes using the pipe.c program first.
- Try to connect the processes in pipe.c so that the file read in the first process is written down a pipe which is read from in the second process.

Adding exec

- When you are confident your pipeline is working correctly, all that is left is to tell each process to exec the appropriate binary.
- exec replaces the current process image with a new process image specified by a binary file name and arguments.
- Once the image is replaced, you have no control over what the process does (which is why it is recommended that you test the pipeline well before this step).

execl()

- There are many different flavors of exec.
  - They give the programmer options as to how he or she would like to specify an executable and its arguments.
- int execl(const char *path, const char *arg1, . . .)
  - Allows you to specify the path to the executable and the arguments to the executable (as you would type at the command line) as a variable number of const char * arguments.
  - Final argument must be (char *) 0

execl example

```
if (pid_1 == 0) {
   /* First Child */
   char cmdbuf[BSIZE];
   bzero(cmdbuf, BSIZE);
   sprintf(cmdbuf, "%s %s -name \'*\'.[ch]", FIND_EXEC, argv[1]);

   /* set up pipes */
   …

   if ( (execl(BASH_EXEC, BASH_EXEC, "-c", cmdbuf, (char *) 0)) < 0) {
     fprintf(stderr, "\nError execing find. ERROR#%d\n", errno);
     return EXIT_FAILURE;
   }
}
```

Finishing Up

- After you have each completed your implementation, compile the finder program and run the test code:

bash> make test

- If the diff line does not produce an error, your implementation is correct.