Operating Systems

06r. Assignment 5 Discussion

Paul Krzyzanowski

Rutgers University

Spring 2015

Assignment 5

- Write a simple shell
  - Read one line: command and arguments
  - Run the command with the given arguments
  - Wait for the command to exit
  - Print the exit code of the command
- You need to support built-in commands
  - cd *dirname*
    Change the current working directory to *dirname*
  - exit *value*
    Exit the shell. Optionally specify a *value* for the exit code

What you need to support

- You need to support built-in commands
  - **`cd dirname`**
    Change the current working directory to *dirname*
  - **`exit value`**
    Exit the shell. Optionally specify a *value* for the exit code
- You need to support pipes
  - Pipe: ability to redirect the output of one program to the input of another program

You do not need to support

- A command that spans multiple lines
- Background processes
- Environment variables
- Multiple commands per line
  - E.g.: `pwd; echo hello; ls /; who`
- Programming constructs
  - E.g., while, for, if, do
- I/O redirection
  - E.g., `ls -l >outfile`
- Any other constructs not specifically mentioned

Understanding pipes

- Guiding philosophy in the design of Unix commands and the Unix shell
  - A set of small, well-defined commands
  - Each command does one thing
  - The output of a command should ideally be in a format that is useful as the input to another command (avoid headers and other junk)
  - Most output is text-based and line-oriented
    - Each line usually represents a complete record or nugget of data

Understanding pipes

- Example: *how many files are in the current directory?*

```
ls | wc -l
```

  - Send the output of *ls* (list files) to *wc -l* (word count: count lines)
  - Counts the number of files in the current directory

- Example: *how many processes is each user running?*

```
ps axu|sort|cut -d ' ' -f1|uniq -c|sort -n
```

  - ps axu: list of processes – first field = user name
  - sort: sort the list alphabetically
  - cut –d ‘ ‘ -f 1: extract the first field of each line, delimiter = space
  - uniq -c: count unique adjacent lines
  - sort -n: the output numerically

Doing the assignment

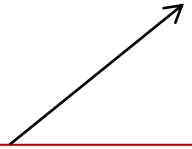
- Develop your code incrementally
  - Write a few lines of code and then test
  - Do not write the entire shell and then wonder why it does not work
- Most of your code will deal with parsing!
  - You must be comfortable with strings in C
- Partition the work
  - You can work in a team of up to five students
  - Get the parsing working on its own
    - Before you add in the system calls
  - Go through the tutorials (see the class *Documents* page)
    - “Playing with processes”
    - “I/O redirection and IPC”
    - Make sure you understand the system calls and can run the demos

Step 1: get a command

- Version 0.00
  - Print a prompt
  - Read a line containing a command
  - Print it (for debugging – you'll remove this later)
  - Repeat
- Print the prompt only if the input is a terminal (not a file)
  - Detect this with *isatty(0)*

```
int showprompt = isatty(0);

if (showprompt) fputs(prompt, stderr);
```



stderr = standard error stream
This is typically the terminal even if you redirected output to a file

Step 2: parse command into tokens

- Parse the command that you just read
  - Create a list of tokens: `char **args`
  - Spaces and tabs separate tokens
  - Tokens may be quoted to include spaces and/or tabs
  - Example:

    ```
    test  "this is a test"   '  hello'
    ```

    will give you a list of

    ```
    { "test", "this is a test", "  hello", 0 }
    ```

  - Terminate each list with a 0 so you know when you reach the end
- Write your own token parser – *gettok* does not handle quotes
  - You should NOT have to call *malloc* and/or copy strings
    - Just parse in place, set pointers to what you need, and set bytes to 0 to mark an end of a string

Step 3: parse a list of commnands

- Create a list of token lists
- For example:

```
ps axu|sort|cut -d ' ' -f1|uniq -c|sort -n
```

- Produces 5 lists:
  - command 1: { “ps”, “axu”, 0 }
  - command 2: { “sort”, 0 }
  - command 3: { “cut”, “-d”, “ “, “-f1”, 0}
  - command 4: { “uniq”, “-c”, 0}
  - command 5: { “sort”, “-n”, 0}

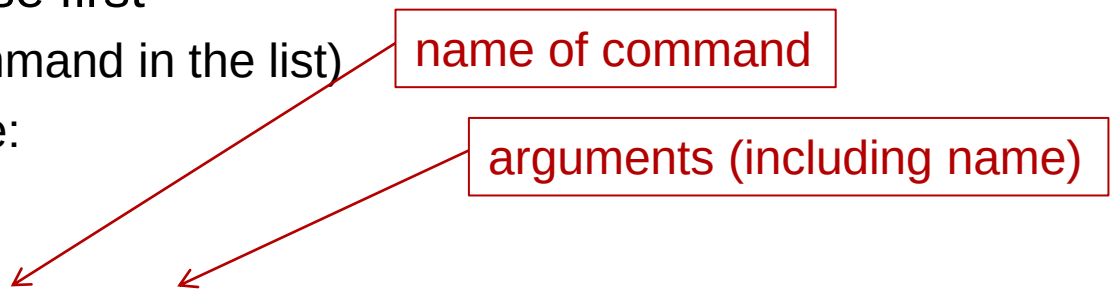
Use an array of pointers to tokens for each command:
e.g., char **args[MAXA];

Use a linked list for the entire pipeline of commands (this is the only place in your code where you may choose to use *malloc*)

- Print these out:
  Make sure you’re capturing all the data.

Step 4: run simple commands

- Now we have a list of commands
- Each command is an array of pointers to strings
- Handle the simple case first
  - No pipe (just one command in the list)
  - Follow the demo code:
    - `fork()`
    - child:
      - call `execvp(cmd->av[0], cmd)`
        where *cmd* is a pointer to the struct that contains your argument list
        The first argument is the name of the command
    - parent:
      - `wait` for the command to exit
      - print the process ID of that command and the error code



Step 5: the *pipe* system call

- The pipe system call creates two open files:

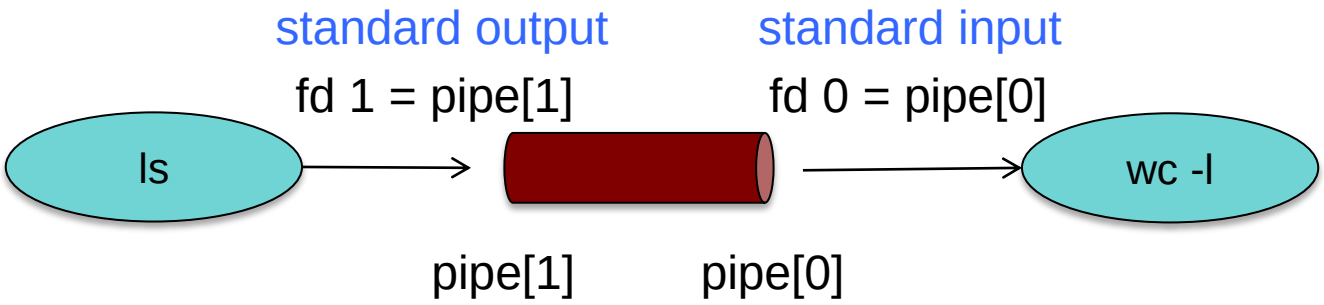
```
int pipe(int fd[2])
```

- Anything written to fd[1] can be read from fd[0]
- These are not files in the file system – just a communication mechanism

Step 5: get pipes working

- A command expects three open files:
  - File descriptor 0 = standard input (normally your keyboard)
  - File descriptor 1 = standard output (normally your terminal window)
  - File descriptor 2 = standard error (normally your terminal window)
- Read the tutorial on I/O redirection using *dup2* and *pipe*

Parent creates a pipe: p[2]
Each child:
Prior to calling *execvp*, overwrite the standard output and standard input



Step 5: get pipes working

- Before calling *exec* to run a command, the child does:

  if the command is getting its input from a pipe (another command)

    Use *dup2* to set the standard input (0) to fd[0] of the pipe

  if there is another command in the pipeline

    Use *dup2* to set the standard output (1) to fd[1] of the next pipe

    (the next command will read from the corresponding fd[0])

  close any ends of the pipe that you don't need

  execvp(cmd->args[0], cmd->args);

Built-in commands

- Built-in commands
  - Processed by the shell directly
  - exit *N*: exit the shell with a exit code of N
  - cd *D*: change the current working directory to D

- For this assignment, you do *NOT* need to support built-in commands inside a pipeline
- Prior to creating a child via *fork*
  - Check the command (argument 0) to see if it is a built-in command
  - Make this process table-driven
    - Declare a table of structs so you can iterate through the table to find the command and corresponding function
    - This keeps your code really short and clean
    - Makes it easier to add new built-in commands

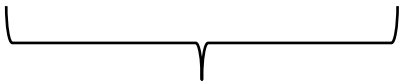
Built-in commands

- Example:

```
struct builtin {
    char *name; /* command name */
    int (*f)();    /* pointer to function */
}
```

- Have each command look like *main(int argc, char**argv)*
  - This makes it easy to turn programs into built-in commands
  - We already parsed out an argument list → count the arguments to get *argc*
- If you the command matches a built-in command, call

```
builtins[i].f(cmd->argc, cmd->args);
```



The function pointer in builtins[i]

The End