Week 3
Shell Extras

Sarah Diesburg
Operating Systems
CS 3430
Environmental Variables

- Gives programs specific information about your environment, such as your execution paths

```
sarah@trogdor:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games
```

- May be set by you or other shell scripts (like .bashrc)

```
sarah@trogdor:~$ export TEST=hello
sarah@trogdor:~$ echo $TEST
hello
```
Environmental Variables

```
char *getenv(const char *name);
```

- Returns value of an environmental variable, NULL if not found
- Overwrites current variable info with value

```
char *setenv(const char *name, const char *value, int overwrite);
```
Environmental Variables

- Important examples
  - $PATH
  - $USER
  - $PWD

- Many shell prompts incorporate these
- The command ‘pwd’ also uses $PWD
Resolving Pathnames?

- Sometimes we don’t pass the full command name to the shell
  - E.g. pass ‘ls’ instead of ‘/bin/ls’
- Something has to translate ‘ls’ into ‘/bin/ls’
- Execvp searches all of the user’s paths stored in the $PATH environmental variable
Finding full pathname for ‘ls’

$PATH=/usr/local/bin:/usr/bin:/bin

- Does /usr/local/bin/ls exist?
  - No
- Does /usr/bin/ls exist?
  - No
- Does /bin/ls exist?
  - Yes!
$PATH

- Where is $PATH stored?

- What happens when we add a ".." to the end of $PATH?
cd

- We are very used to using cd, but it is not a real command.
  - Try ‘which cd’ at the prompt – there isn’t one!

- The command cd is actually a built-in command of the shell
int chdir (const char *path)

- Changes the working directory
- Also need to update the $PWD so that it matches
  - Otherwise the pwd command will be strange
  - Use setenv() to update $PWD
cd Behavior

- **cd** with no arguments
  - Reverts the present working directory to $HOME.
- **cd** with ‘.’
  - PWD does not change.
- **cd** with ‘..’
  - PWD changes to the parent.
- **cd** <dir>
  - If DIR is found and DIR is a directory, change the PWD to DIR
  - If not, signal that DIR does not exist
cd Implementation Hints

- Look at the following:
  - `setenv`
  - `getcwd`
  - `chdir`
  - `pwd`

- Another hint: use the man pages to see what they do
  - `$> man getcwd`
The following slides give you an under-the-hood explanation of pipes and redirection in a shell.

- What does pipe | do?
- What does < > do?
Redirection

- Redirection of stdin and stdout happen by placing a redirection character between commands
  - `[command] < [in_file]`
  - `[command] > [out_file]`
  - `[command] < [in_file] > [out_file]`
Input Redirection

- [command] < [in_file]
- The command now takes input from in_file instead of stdin
- Examples
  - cat < file1.txt
  - grep hamburger < menu.txt
How to Implement Redirection

- We need to duplicate standard I/O using the ‘dup’ command

```c
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- Dup makes a duplicate of the oldfd file descriptor and returns the new file descriptor
Steps for Input Redirection

1. Open the input file
2. Close stdin
3. Issue dup() on the input file
   - dup will automatically choose lowest closed file descriptor – stdin
   - Input file and stdin are now the same file descriptor
4. Close input file
Input Redirection Visual

fork()

file descriptors
0 stdin
1 stdout
2 stderr

shell

file descriptors
0 stdin
1 stdout
2 stderr

shell
Input Redirection Visual

```
open(input_file, O_RDONLY)
```

file descriptors
0 stdin
1 stdout
2 stderr

shell

file descriptors
0 stdin
1 stdout
2 stderr

shell
Input Redirection Visual

file descriptors
0 stdin
1 stdout
2 stderr

shell

open(input_file, O_RDONLY)

Can be any number > 2…

file descriptors
0 stdin
1 stdout
2 stderr
3 input_file

shell
Input Redirection Visual

file descriptors
0 stdin
1 stdout
2 stderr

shell

file descriptors
0 stdin
1 stdout
2 stderr
3 input_file

close(0)

shell
Input Redirection Visual

file descriptors
0 stdin
1 stdout
2 stderr

shell

dup(3)

file descriptors
0
1 stdout
2 stderr
3 input_file

shell
Input Redirection Visual

file descriptors
0 stdin
1 stdout
2 stderr

file descriptors
0 input_file
1 stdout
2 stderr
3 input_file

dup(3)

shell

shell
Input Redirection Visual

shell

file descriptors
0 stdin
1 stdout
2 stderr

shell

file descriptors
0 input_file
1 stdout
2 stderr
3 input_file

close(3)
Output Redirection

- [command] > [out_file]
- Prints the output from command to out_file instead of stdout
- Example
  - `echo hello > file1.txt`
Steps for Output Redirection

1. Open the output file
2. Close stdout
3. Issue dup() on the output file
   - dup will automatically choose lowest closed file descriptor – stdout
   - Input file and stdout are now the same file descriptor
4. Close output file
Output Redirection Visual

fork()

shell

file descriptors
0 stdin
1 stdout
2 stderr

shell

file descriptors
0 stdin
1 stdout
2 stderr
Output Redirection Visual

open(output_file, O_RDWR | O_CREAT | O_TRUNC)

directory

file descriptors
0 stdin
1 stdout
2 stderr

shell

file descriptors
0 stdin
1 stdout
2 stderr

shell
Output Redirection Visual

file descriptors
0 stdin
1 stdout
2 stderr

shell

open(output_file, O_RDWR | O_CREAT | O_TRUNC)

file descriptors
0 stdin
1 stdout
2 stderr
3 output_file

shell
Output Redirection Visual

file descriptors
0 stdin
1 stdout
2 stderr

shell

close(1)

file descriptors
0 stdin
1 stdout
2 stderr
3 output_file

shell
Output Redirection Visual

file descriptors
0 stdin
1 stdout
2 stderr

shell

dup(3)

file descriptors
0 stdin
1
2 stderr
3 output_file

shell
Output Redirection Visual

file descriptors
0 stdin
1 stdout
2 stderr

shell

dup(3)

file descriptors
0 stdin
1 output_file
2 stderr
3 output_file

shell
Output Redirection Visual

file descriptors
0 stdin
1 stdout
2 stderr

file descriptors
0 stdin
1 output_file
2 stderr
3 output_file

shell

close(3)
Combination Redirection

- `[command] < [in_file] > [out_file]`
- The command takes input from the in_file instead of stdin and prints output to the out_file instead of stdout

Example
- `cat < file1.txt > duplicate.txt`
Pipes

- One program can receive output from another without an explicit temporary file:

  `command1 | command2`

- Same as:

  `command1 > tmpfile`
  `command2 < tmpfile`
  `rm tmpfile`
Pipes

Since two commands will be run, we will need to create two children

We also need to create an I/O “pipe” between the children processes
**Pipes**

- Use the system call ‘pipe’

  ```c
  int pipe(int filedes[2]);
  ```

- Creates a pair of file descriptors for pipe I/O and places them in filedes[]

- Returns 0 on success
Pipe Example

- `$> ps | grep bash`
  - Finds your bash process
- `$> ps aux | grep bash`
  - Finds everyone’s bash process

So what do we think grep does?

How can we find all processes owned by us?
Pipe vs Redirection

So, when would we want to use a redirection over a pipe?

- Pipe over redirection?
Any Other Questions?