



Basic OS Programming Abstractions (and Lab 1 Overview)

Don Porter

Portions courtesy Kevin Jeffay



Recap

- We've introduced the idea of a process as a container for a running program
- This lecture: Introduce key OS APIs for a process
 - Some may be familiar from lab 0
 - Some will help with lab 2



Lab 1: A (Not So) Simple Shell

- Lab 1: Parsing for a shell
 - You will extend in lab 2
- I'm giving you some boilerplate code that does basics
- My goal: Get some experience using process APIs
 - Most of what you will need discussed in this lecture
- You will incrementally improve the shell



Tasks

- Turn input into commands; execute those commands
 - Support PATH variables
- Be able to change directories
- Print the working directory at the command line
- Add debugging support
- Add scripting support
- Pipe indirection: `<`, `>`, and `|`
- **goheels** – draw an ASCII art Tar Heel

Significant work – start early!



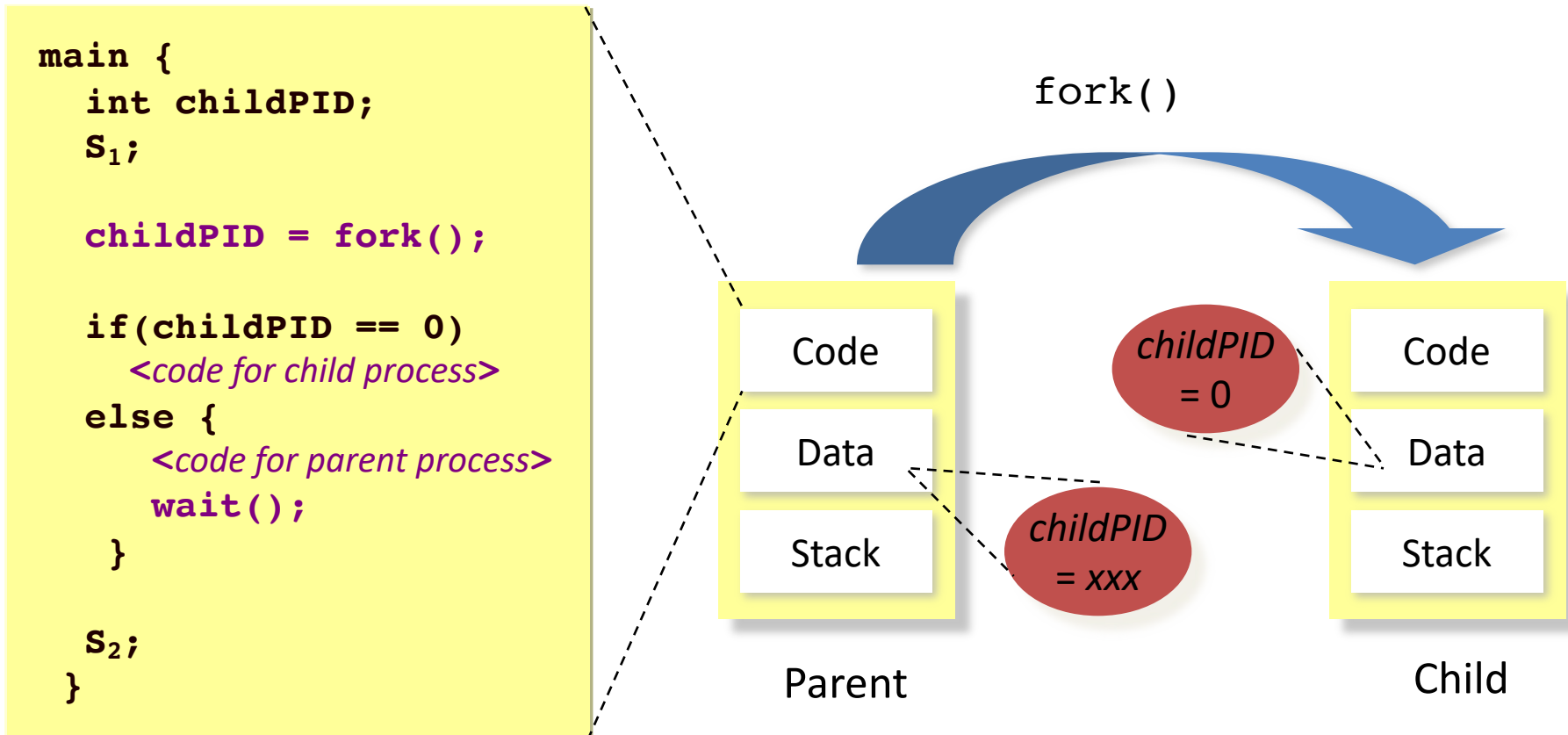
Outline

- Fork recap
- Files and File Handles
- Inheritance
- Pipes
- Sockets
- Signals
- Synthesis Example: The Shell



Process Creation: fork/join in Linux

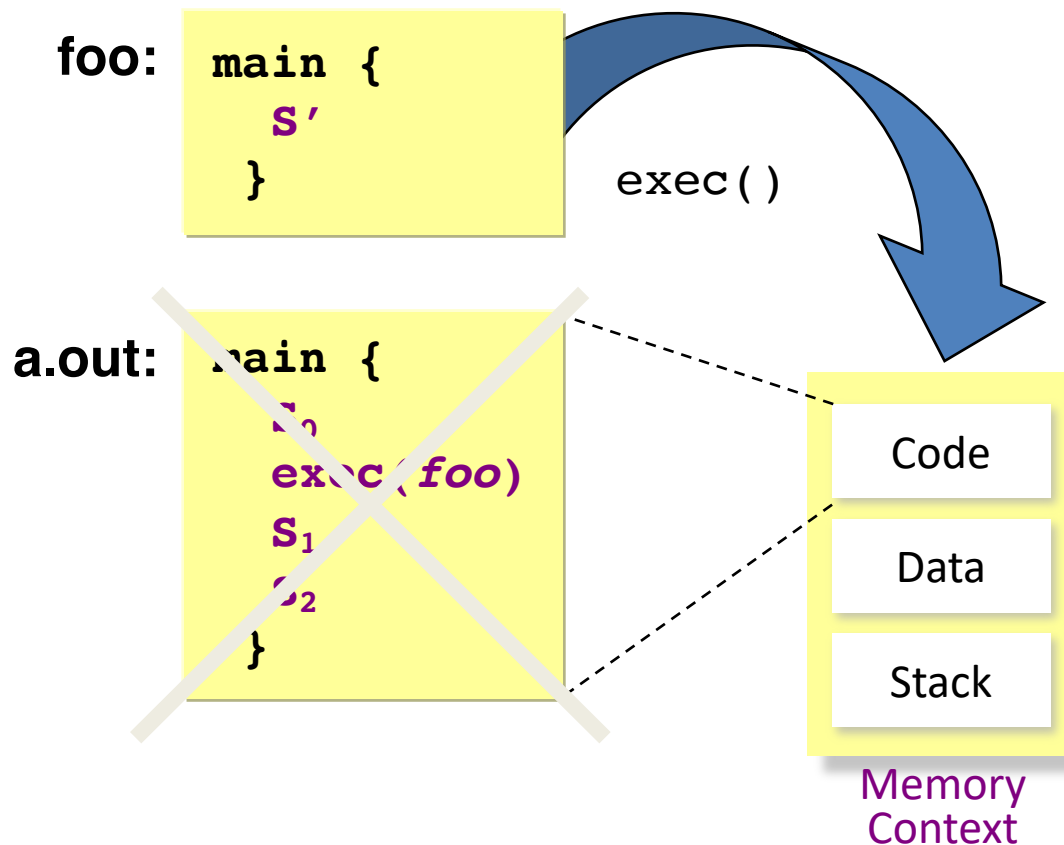
- The execution context for the child process is a *copy* of the parent's context at the time of the call





Process Creation: exec in Linux

- *exec* allows a process to replace itself with another program
 - (The contents of another binary file)

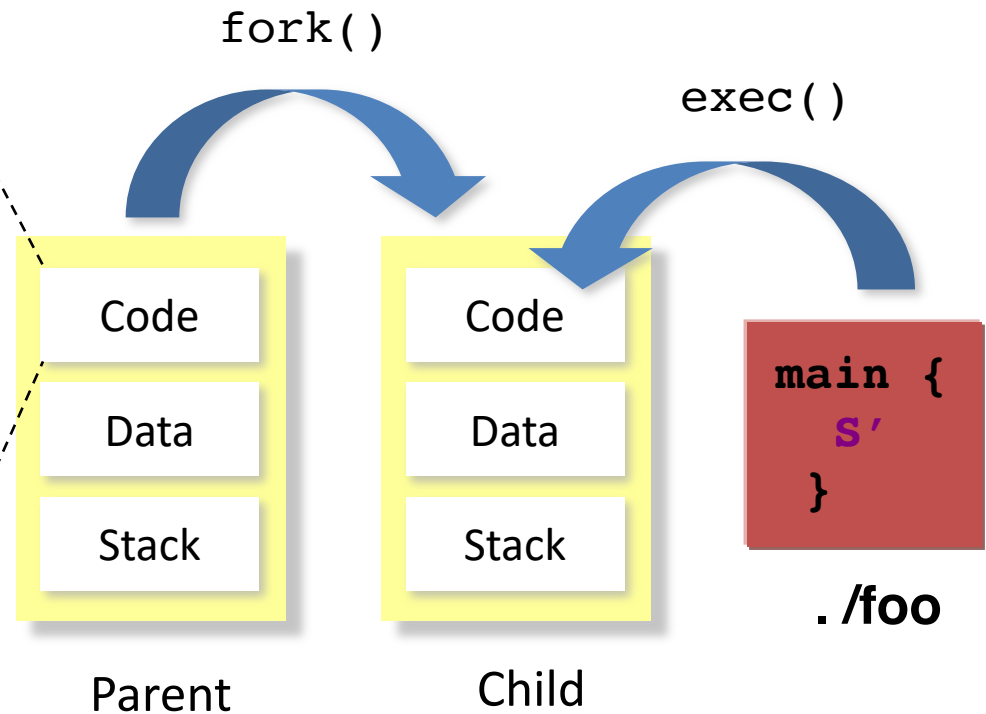




Process Creation: Abstract fork in Linux

- Common case: *fork* followed by an *exec*

```
main {  
  int childPID;  
  S1;  
  
  childPID = fork();  
  
  if(childPID == 0)  
    exec(filename)  
  else {  
    <code for parent process>  
    wait();  
  }  
  
  S2;  
}
```





2 Ways to Refer to a File

- Path, or hierarchical name, of the file
 - Absolute: “/home/porter/foo.txt”
 - Starts at system root
 - Relative: “foo.txt”
 - Assumes file is in the program’s current working directory
- Handle to an open file
 - Handle includes a cursor (offset into the file)



Path-based calls

- Functions that operate on the directory tree
 - Rename, unlink (delete), chmod (change permissions), etc.
- Open – creates a handle to a file
 - `int open (char *path, int flags, mode_t mode);`
 - Flags include `O_RDONLY`, `O_RDWR`, `O_WRONLY`
 - Permissions are generally checked only at open
 - `Opendir` – variant for a directory



Handle-based calls

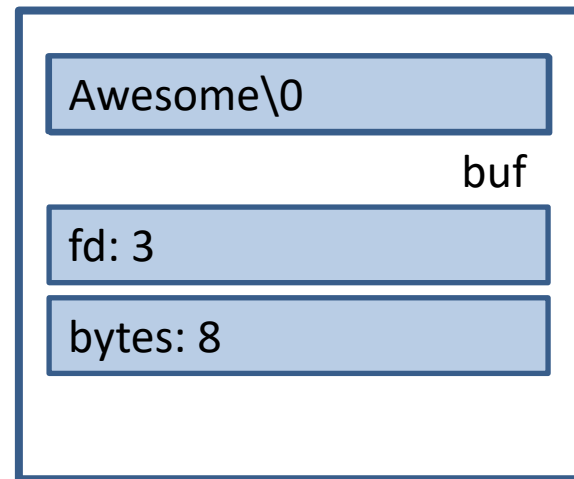
- `ssize_t read (int fd, void *buf, size_t count)`
 - Fd is the handle
 - Buf is a user-provided buffer to receive count bytes of the file
 - Returns how many bytes read
- `ssize_t write(int fd, void *buf, size_t count)`
 - Same idea, other direction
- `int close (int fd)`
 - Close an open file
- `int lseek(int fd, size_t offset, int flags)`
 - Change the cursor position



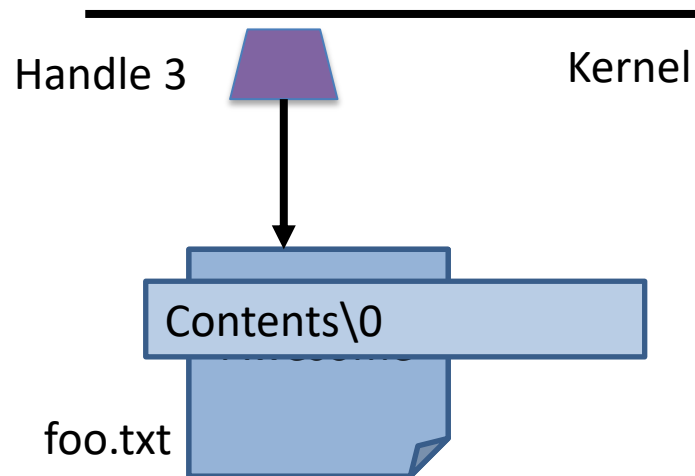
Example

PC

```
char buf[9];  
int fd = open ("foo.txt", O_RDWR);  
ssize_t bytes = read(fd, buf, 8);  
if (bytes != 8) // handle the error  
lseek(3, 0, SEEK_SET); //set cursor  
memcpy(buf, "Awesome", 7);  
buf[7] = '\\0';  
bytes = write(fd, buf, 8);  
if (bytes != 8) // error  
close(fd);
```



User-level stack





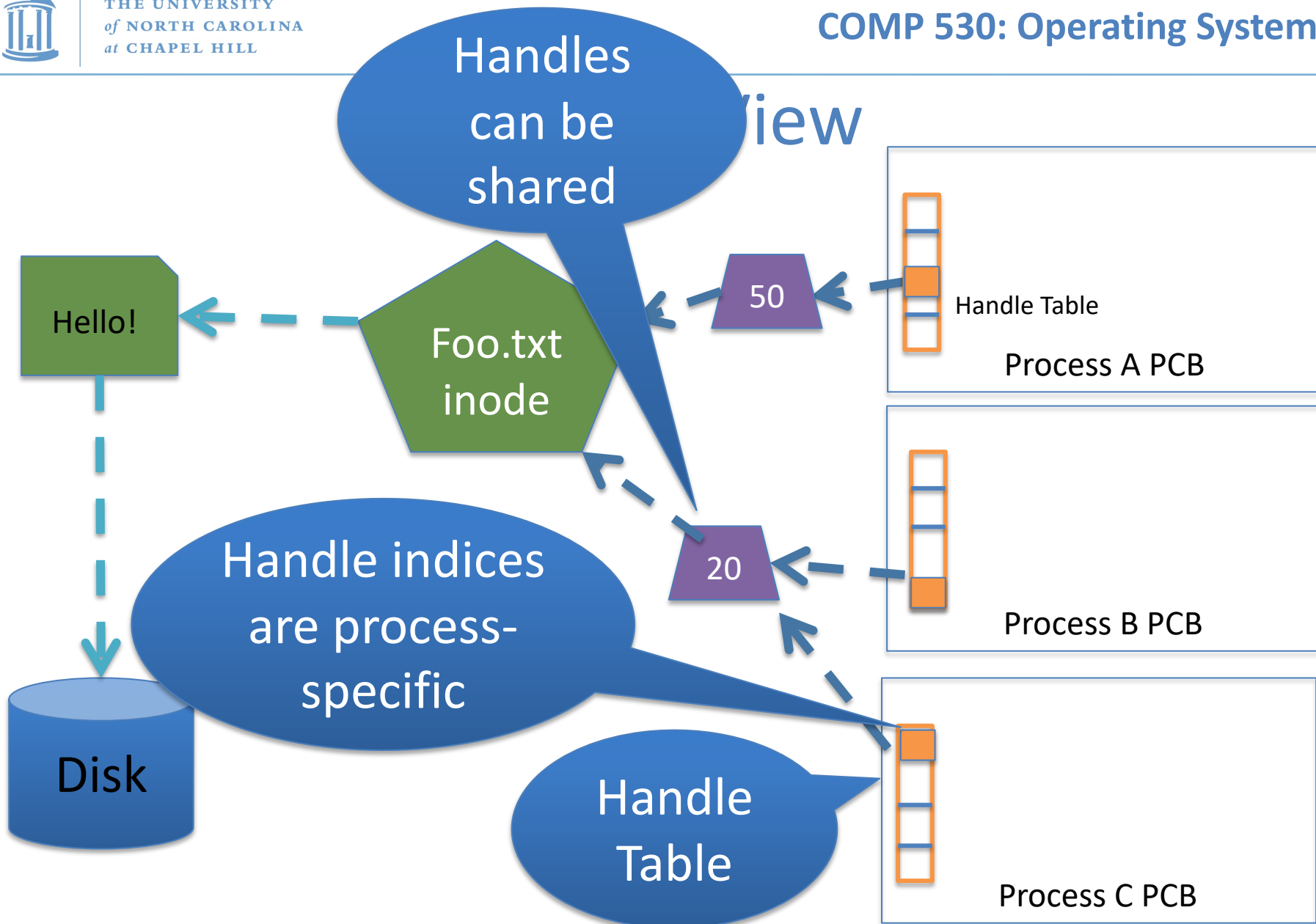
Why handles?

- Handles in Unix/Linux serve three purposes:
 1. Track the offset of last read/write
 - Alternative: Application explicitly passes offset
 2. Cache the access check from `open()`
 3. Hold a reference to a file
 - Unix idiom: Once a file is open, you can access the contents as long as there is an open handle --- even if the file is deleted from the directory



But what is a handle?

- A reference to an open file or other OS object
 - For files, this includes a cursor into the file
- In the application, a handle is just an integer
 - This is an offset into an OS-managed table





Handle Recap

- Every process has a table of pointers to kernel handle objects
 - E.g., a file handle includes the offset into the file and a pointer to the kernel-internal file representation (inode)
- Applications can't directly read these pointers
 - Kernel memory is protected
 - Instead, make system calls with the indices into this table
 - Index is commonly called a handle



Rearranging the table

- The OS picks which index to use for a new handle
- An application explicitly copy an entry to a specific index with `dup2(old, new)`
 - Be careful if new is already in use...



Other useful handle APIs

- `mmap()` – can map part or all of a file into memory
- `seek()` – adjust the cursor position of a file
 - Like rewinding a cassette tape





Outline

- Files and File Handles
- Inheritance
- Pipes
- Sockets
- Signals
- Synthesis Example: The Shell



Inheritance

- By default, a child process gets a reference to every handle the parent has open
 - Very convenient
 - Also a security issue: may accidentally pass something the program shouldn't
- Between `fork()` and `exec()`, the parent has a chance to clean up handles it doesn't want to pass on
 - See also `CLOSE_ON_EXEC` flag



Standard in, out, error

- Handles 0, 1, and 2 are special by convention
 - 0: standard input
 - 1: standard output
 - 2: standard error (output)
- Command-line programs use this convention
 - Parent program (shell) is responsible to use `open/close/dup2` to set these handles appropriately between `fork()` and `exec()`



Example

```
int pid = fork();  
if (pid == 0) {  
    int input = open ("in.txt",  
                     O_RDONLY);  
    dup2 (input, 0);  
    exec ("grep", "quack");  
}  
//...
```



Outline

- Files and File Handles
- Inheritance
- Pipes
- Sockets
- Signals
- Synthesis Example: The Shell



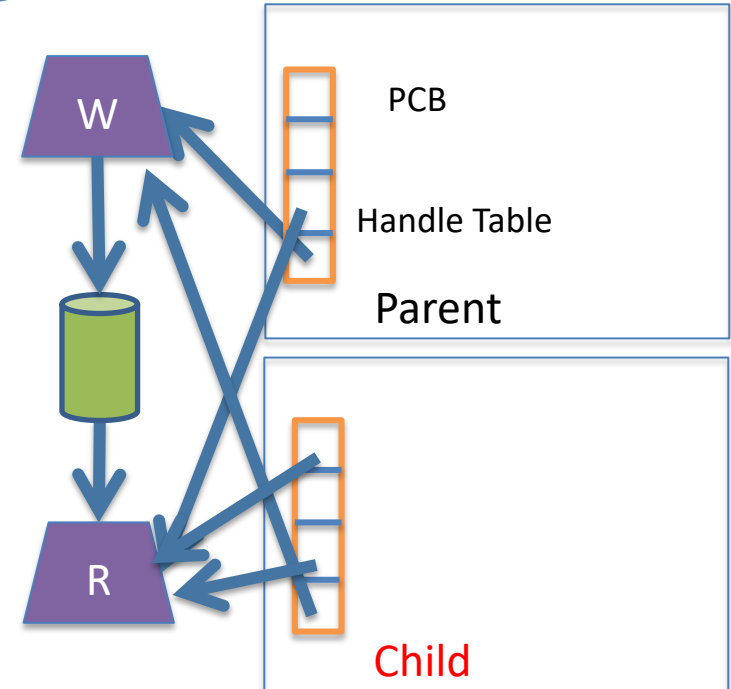
Pipes

- FIFO stream of bytes between two processes
- Read and write like a file handle
 - But not anywhere in the hierarchical file system
 - And not persistent
 - And no cursor or seek()-ing
 - Actually, 2 handles: a read handle and a write handle
- Primarily used for parent/child communication
 - Parent creates a pipe, child inherits it



Example

```
PC → int pipe_fd[2];  
      int rv = pipe(pipe_fd);  
PC → int pid = fork();  
      if (pid == 0) {  
          close(pipe_fd[1]);  
          dup2(pipe_fd[0], 0);  
          close(pipe_fd[0]);  
          exec("grep", "quack");  
      } else {  
          close(pipe_fd[0]);  
          ...  
      }
```



Goal: Create a pipe; parent writes, child reads



Sockets

- Similar to pipes, except for network connections
- Setup and connection management is a bit trickier
 - A topic for another day (or class)



Select

- What if I want to block until one of several handles has data ready to read?
- Read will block on one handle, but perhaps miss data on a second...
- Select will block a process until a handle has data available
 - Useful for applications that use pipes, sockets, etc.



Outline

- Files and File Handles
- Inheritance
- Pipes
- Sockets
- Signals
- Synthesis Example: The Shell



Signals

- Similar concept to an application-level interrupt
 - Unix-specific (more on Windows later)
- Each signal has a number assigned by convention
 - Just like interrupts
- Application specifies a handler for each signal
 - OS provides default
- If a signal is received, control jumps to the handler
 - If process survives, control returns back to application



Signals, cont.

- Can occur for:
 - Exceptions: divide by zero, null pointer, etc.
 - IPC: Application-defined signals (USR1, USR2)
 - Control process execution (KILL, STOP, CONT)
- Send a signal using `kill(pid, signo)`
 - Killing an errant program is common, but you can also send a non-lethal signal using `kill()`
- Use `signal()` or `sigaction()` to set the handler for a signal



How signals work

- Although signals appear to be delivered immediately...
 - They are actually delivered lazily...
 - Whenever the OS happens to be returning to the process from an interrupt, system call, etc.
- So if I signal another process, the other process may not receive it until it is scheduled again
- Does this matter?



More details

- When a process receives a signal, it is added to a pending mask of pending signals
 - Stored in PCB
- Just before scheduling a process, the kernel checks if there are any pending signals
 - If so, return to the appropriate handler
 - Save the original register state for later
 - When handler is done, call `sigreturn()` system call
 - Then resume execution



Meta-lesson

- Laziness rules!
 - Not on homework
 - But in system design
- Procrastinating on work in the system often reduces overall effort
 - Signals: Why context switch immediately when it will happen soon enough?



Language Exceptions

- Signals are the underlying mechanism for Exceptions and catch blocks
- JVM or other runtime system sets signal handlers
 - Signal handler causes execution to jump to the catch block



Windows comparison

- Exceptions have specific upcalls from the kernel to ntdll
- IPC is done using Events
 - Shared between processes
 - Handle in table
 - No data, only 2 states: set and clear
 - Several variants: e.g., auto-clear after checking the state



Outline

- Files and File Handles
- Inheritance
- Pipes
- Sockets
- Signals
- Synthesis Example: The Shell



Shell Recap

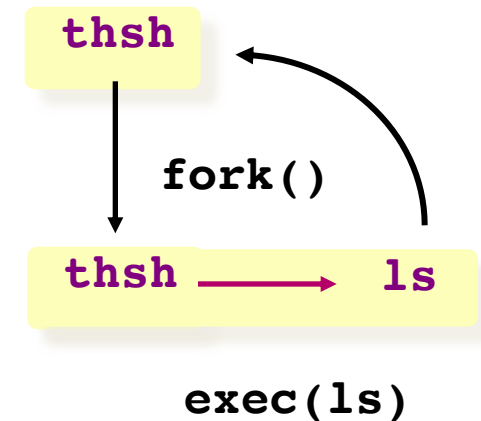
- Almost all ‘commands’ are really binaries
 - /bin/l`s`
- Key abstraction: Redirection over pipes
 - ‘>’, ‘<’, and ‘|’ implemented by the shell itself



Shell Example

- Ex: `ls | grep foo`
- Shell pseudocode:

```
while(EOF != read_input) {  
    parse_input();  
    // Sets up chain of pipes  
    // Forks and exec's 'ls' and 'grep' separately  
    // Wait on output from 'grep', print to console  
    // print console prompt  
}
```





Lab 1 Overview

- C programming on Linux refresher
- Parser for your shell (Lab 1)



Shells

- Shell: aka the command prompt
- At a high level:

```
while (more input) {  
    read a line of input  
    parse the line into a command  
    if valid command: execute it  
}
```

We will give you this

Lab 1

Lab 2



Detour: Environment Variables

- Nearly all shell commands are actually binary files
 - Very few commands actually implemented in the shell
 - A few built-ins that change the shell itself (exit, cd)
- Example: **ls** is actually in **/bin/ls**
 - For fun, play with **which**, as in **which ls**
- So where to look for a given command?
 - Note that we want some flexibility system-to-system
- Idea: dynamically set a variable that controls which directories to search



Environment Variables

- A set of key-value pairs
 - Passed to main() as a third argument
 - Often ignored by programmers
- Serves many different purposes
- For Lab 1, we need to look at PATH
 - By convention, a single, colon-delimited set of prefixes
- Example:

```
/usr/local/sbin:/usr/local/bin:/usr/s  
bin:/usr/bin:/sbin:/bin
```



PATH in a shell

- If my PATH is

```
/usr/local/sbin:/usr/local/bin:/usr/sbin  
:/usr/bin:/sbin:/bin
```

- Then, for a given command (ls), the shell will check, in order, until found:

```
/usr/local/sbin/ls
```

```
/usr/local/bin/ls
```

```
/usr/sbin/ls
```

```
/usr/bin/ls
```

```
/sbin/ls
```

```
/bin/ls
```



Lab 1, Exercise 1

- Your first job will be to write parsing code that takes in a colon-delimited set of prefixes, and to create a table of prefixes to try in future commands
 - See `path_table` in `jobs.c`
 - We wrote a test harness `test_env.c`

```
$ PATH=/foo:/bar ./test_env
```

```
===== Begin Path Table =====
```

```
Prefix 0: [/foo]
```

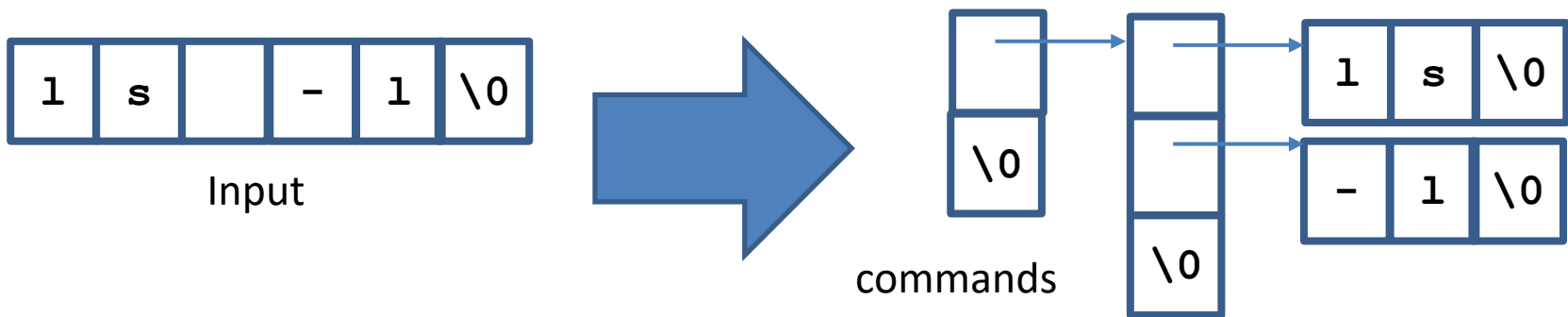
```
Prefix 1: [/bar]
```

```
===== End Path Table =====
```



Ex 2: Parsing commands

- A typical shell command includes a main binary (e.g., 'ls')
 - and 0+ whitespace-separated arguments (e.g., '-l')
 - and possibly extra whitespace
- You will get this as a single character array
- Your job is to break this up into individual 'tokens'





Pipelines

- A shell can compose multiple commands using pipelines
 - Key idea: standard output of one command becomes standard input of next
- Example: **ls | wc -l**
 - List a directory (ls) – send listing output to a wordcount utility (wc) to count how many entries in directory
- The vertical bar (|) is a special character
 - May not appear in any other valid commands
 - Does not need whitespace: **ls|wc -l** is valid



parse.c:parse_line()

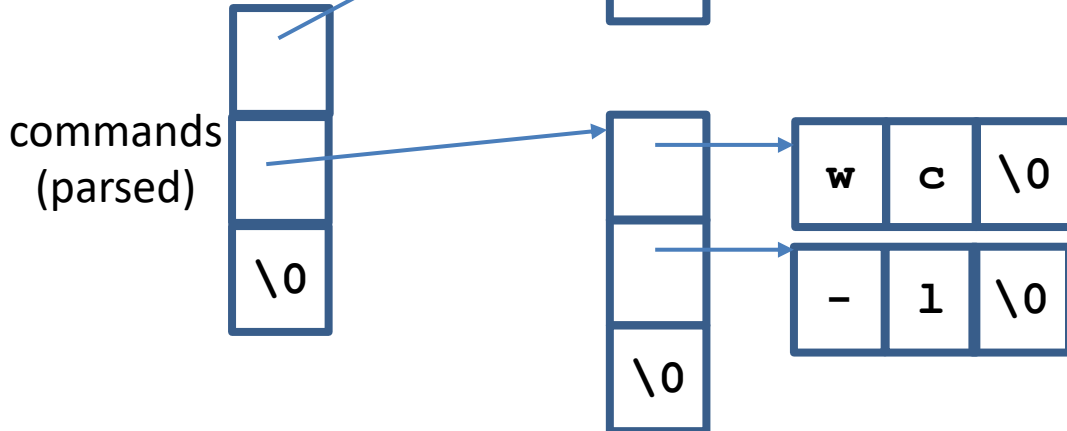
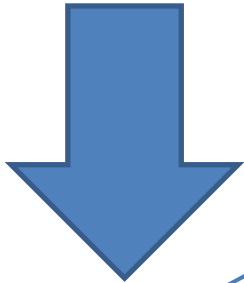
- The workhorse for lab 1 (and 2)
- Takes in a line of input, outputs a 2-D array
- First dimension: one entry per pipeline stage
 - Simple commands just have one entry
- Second dimension: one entry per command token



How to parse a pipeline?



Input





Other special cases

- Comments – anything past a ‘#’ character
- File redirection - sets standard input/output to a file
 - Example: `ls > mydir.txt`
 - Saves the output of ls into a file
 - Example: `wc -l < mydir.txt`
 - Sends the contents of mydir.txt into wc as standard input
- Built-in commands (see builtin.c)
 - For now, you just need to recognize them and call the appropriate handler function



Working on Homework Assignments

- Use the same learncli211 container as lab 0



Checking out the starter code

- Once you have a github account registered
 - Make sure you accept the invite:
 - Click <https://github.com/comp530-f23>
- Click the link in the homework to create a private repo
- Then, on your machine or classroom (substituting your team for 'team-don' – see the green clone button):

```
git clone git@github.com:comp530-f23/thsh-team-don.git
```



Submitting homework

- We will be using gradescope to submit and autograde the homework
 - Challenge problems and late hours done manually
 - Submit challenges separately
- Ideally, use github connection to directly submit
- Feel free to try early to catch issues with grading



A note on Lab 2



thsh

wait()



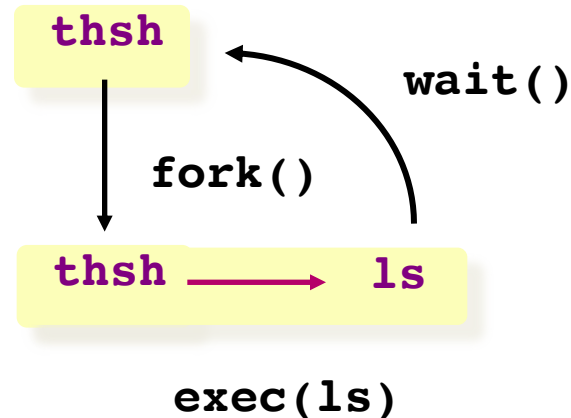
- You assi
- If yo
- You
 - Z
 - N
 - This means no one can launch a shell to kill the zombies!

is

S!!
el



A note on Lab 1



- Be safe! Limit the number of processes you can create
 - add the command “*limit maxproc 10*” to the file *~/.cshrc*
 - (remember to delete this line at the end of the course!)
- Periodically check for and KILL! zombie processes
 - *ps -ef | egrep -e PID -e YOUR-LOGIN-NAME*
 - *kill pid-number*
- Read the HW handout carefully for zombie-hunting details!



What about Ctrl-Z?

- Shell really uses `select()` to listen for new keystrokes
 - (while also listening for output from subprocess)
- Special keystrokes are intercepted, generate signals
 - Shell needs to keep its own “scheduler” for background processes
 - Assigned simple numbers like 1, 2, 3
- ‘fg 3’ causes shell to send a `SIGCONT` to suspended child

- `Ctrl+C` implemented using `SIGKILL`



Other hints

- `Splice()`, `tee()`, and similar calls are useful for connecting pipes together
 - Avoids copying data into and out-of application



Collaboration Policy Reminder

- You can work alone or as part of a team
 - Must be the same as lab 1; may change starting in lab 2
 - Every line of code handed in must be written by one of the pair (or the boilerplate)
 - No sharing code with other groups
 - No code from Internet
 - Any other collaboration must be acknowledged in writing
 - High-level discussion is ok (no code)
- See written assignment and syllabus for more details

Not following these rules is an Honor Code violation



Summary

- Understand how handle tables work
 - Survey basic APIs
- Understand signaling abstraction
 - Intuition of how signals are delivered
- Be prepared to start writing your shell in lab 2!