

CS 3214: Project 1

The Customizable Shell

Help Session:
Friday February 11, 2022 5:00 PM

Justin Vita <justv@vt.edu>

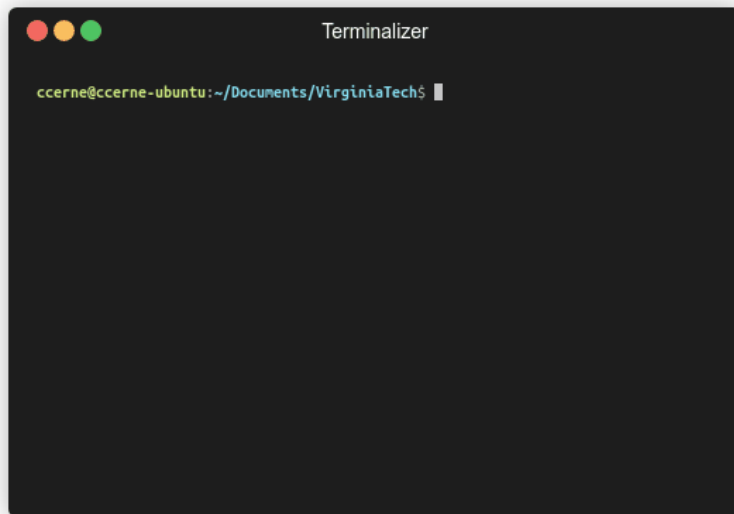
Topics

- Shell Concepts
- Project Overview / Logistics
- Version Control (Git)
- Debugging (GDB)
- Advice
- Q & A

Shell Concepts

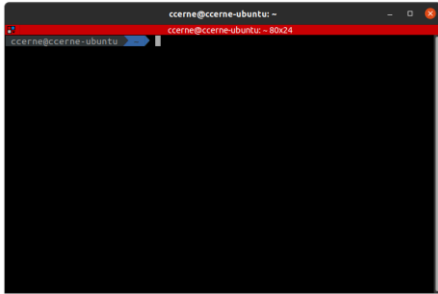
What is a shell?

- Command Interpreter
 - Reads user input and executes user requests
 - Not to be confused with a “Terminal” (next slide explains distinction)



Terminal vs Shell

Terminal (the front-end of our shell)



Examples: gnome-terminal, terminator, Terminal.app (macOS) etc.

The 80s called, they want their Terminal back!

Shell (an executable with no GUI)

```
ccerne@ccerne-ubuntu:~/Documents$ ls -l
ls -l
total 16
drwxrwxr-x 7 ccerne ccerne 4096 Aug 23 10:37 CTF
drwxrwxr-x 6 ccerne ccerne 4096 Sep 11 21:42 Programming
drwxrwxr-x 5 ccerne ccerne 4096 Sep  1 16:56 Programs
drwxrwxr-x 5 ccerne ccerne 4096 Sep 13 21:19 VirginiaTech
ccerne@ccerne-ubuntu:~/Documents$ echo $SHELL
/usr/bin/zsh
```

This terminal is running zsh, a shell

Behind the Scenes

```
$ echo 'Welcome to Systems!'
```



f
o
r
k



```
Welcome to Systems!
```

FOUR STEPS for non-built-in

1. Shell waits for user input
2. Shell interprets command
3. Forks a process
4. If it's a foreground parent waits for child to finish. Else, parent repeats the process again.
 - Child executes the command

Additional Features for the Shell (where you come in)

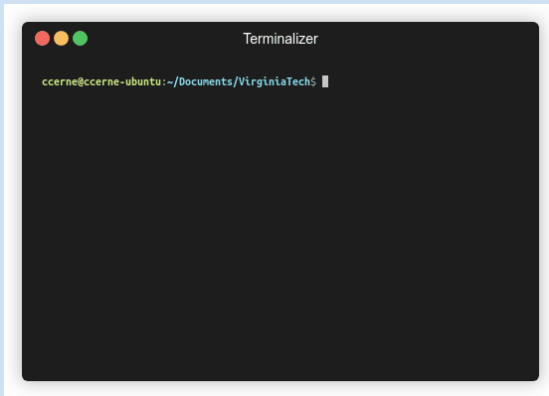
- Foreground / Background Processes
- Process Groups
- Built-in Commands
- I/O Piping
- I/O Redirection
- Signal Handling

Foreground / Background Processes

- The shell can fork processes into the foreground or background

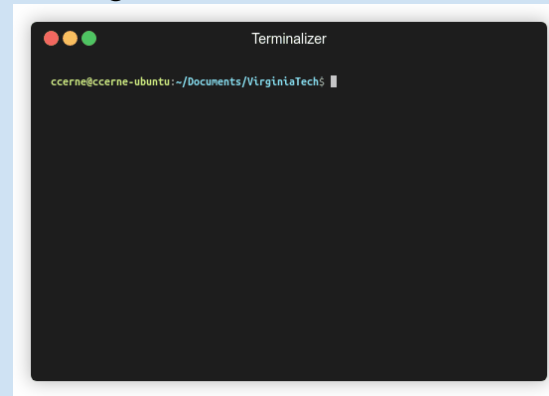
Foreground

- Only one foreground process group at a time
- Have access to the terminal



Background

- Does not have terminal access
- Using '&' sends command to background to run



Process Groups

- Each Job is its own process group
 - Each command within a Job should have the same PGID
 - Two methodologies of creating new processes:
 - Fork
 - `posix_spawn`
- Jobs are deleted when they are completed
 - Be careful not to delete a job prematurely
 - See the comment above `wait_for_job()`

```
<justv@cottonwood justv>$ sleep 20 | sleep 20 | sleep 20 &
```

```
<justv@cottonwood justv>$ ps xj | head -n 1; ps xj | tail -n 6
```

PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND
1357688	1363886	1363886	1357688	pts/0	1365438	S	24908	0:00	/home/courses/cs3214/bin/cush-gback
1363886	1365308	1365308	1357688	pts/0	1365438	S	24908	0:00	sleep 20
1363886	1365309	1365308	1357688	pts/0	1365438	S	24908	0:00	sleep 20
1363886	1365310	1365308	1357688	pts/0	1365438	S	24908	0:00	sleep 20

Notice the PID and PGID!

POSIX Spawn

- Replaces `fork()` + `exec()` entirely
- Code is “linear” rather than handling multiple processes in if-else statements
- Example: [posix_spawn\(3\) - Linux manual page \(man7.org\)](http://man7.org/linux/man-pages/posix_spawn(3).html)

Note: You need to include “`spawn.h`” in your `cush.c` to use these functions. The file is located in the `posix_spawn` directory. Also be sure to use the Makefile and compile `posix_spawn`.

fork() + exec()

```
if (fork() == 0) {
    //child stuff

    execvp(/* program arguments */);
}
else {
    //parentstuff
}
```

posix_spawn()

```
posix_spawn_file_actions_t child_file_attr;
posix_spawnattr_t child_spawn_attr;

posix_spawnattr_int(&child_file_attr);
posix_spawn_file_actions_init(&child_file_attr);

// setup for attributes

posix_spawn(&pid, /*program*/, &child_file_attr,
&child_spawn_attr, /*program arguments*/, environ)
```

We recommend using `posix_spawn()` for this project, but it is not required.

POSIX Spawn Attributes

- Process Groups - `posix_spawnattr_getpgroup ()`
- Terminal Control - `posix_spawnattr_tcsetpgrp_np()`
- Piping - `posix_spawn_file_actions_adddup2()`

More listed on both the spec and `<spawn.h>`

Built-in Commands

- Commands that are defined within the program by you
 - No need to fork off and execute an external program
- Required Built-In Commands for your shell:
 - kill - kills a process
 - jobs - displays a list of jobs
 - stop - stops a process
 - fg - sends a process to foreground
 - bg - sends a process to background
- Two additional built-ins / functionality extenders also required

Built-ins Behind the Scenes

```
$ jobs
```



```
[1]+  Stopped  vim
[2]-  Running  sleep 20 &
```

FOUR STEPS for *built-in*

1. Shell waits for user input
2. Shell realizes this is a built in command
3. Shell executes built-in (no forking)
4. After execution, shell repeats

I/O Piping

```
ls -l | grep *.txt | wc
```



```
ls -l
```

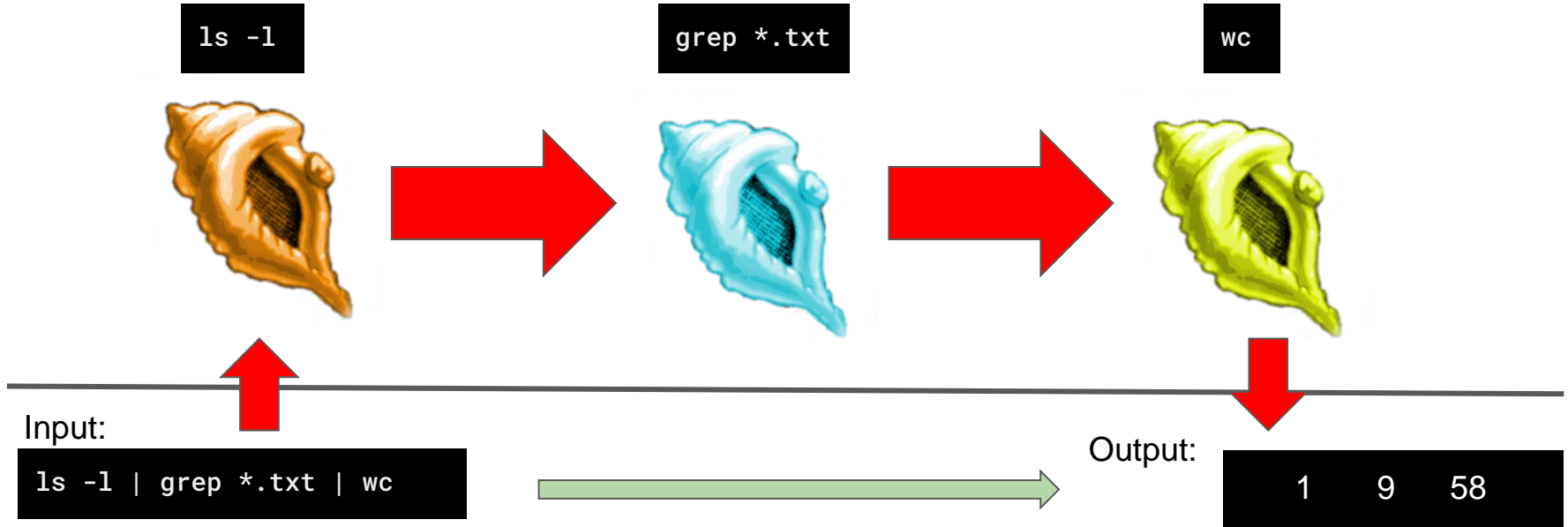
```
grep *.txt
```

```
wc
```

- The Shell will fork off a child process to execute each command in a pipeline
- But since this is a pipeline of commands, we'll also need to wire STDIN and STDOUT for each process....

I/O Piping

- Processes will wait on previous process, final process outputs to terminal
- STDIN and STDOUT for processes are joined to create the pipeline

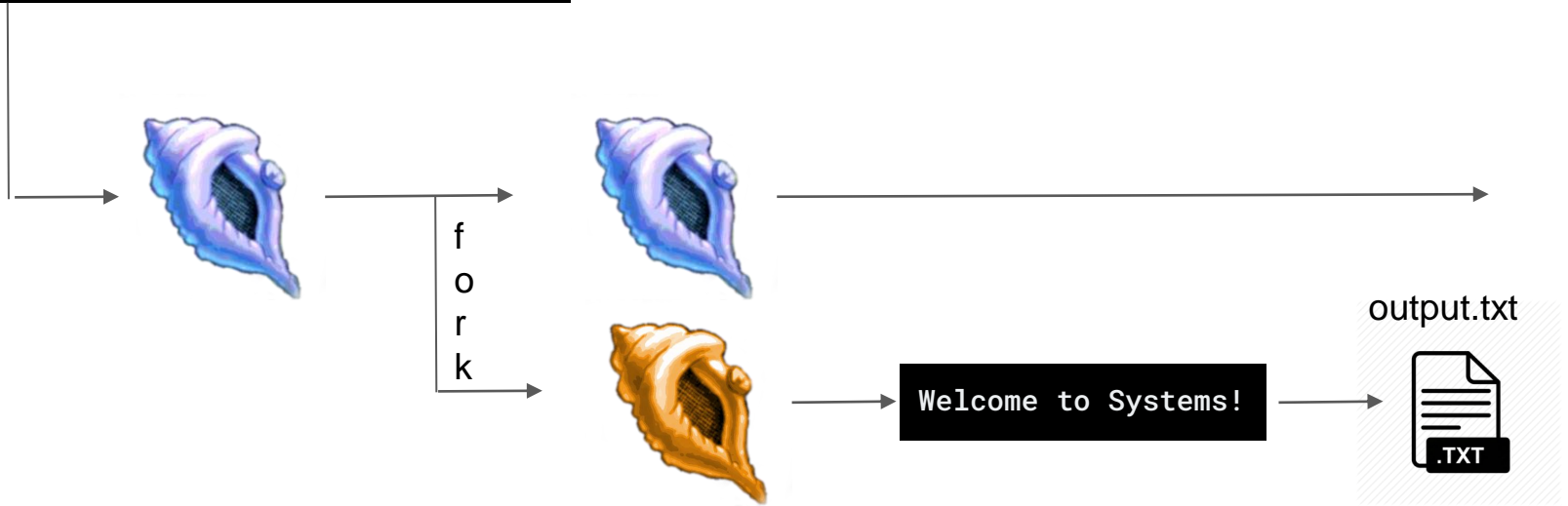


I/O Redirection

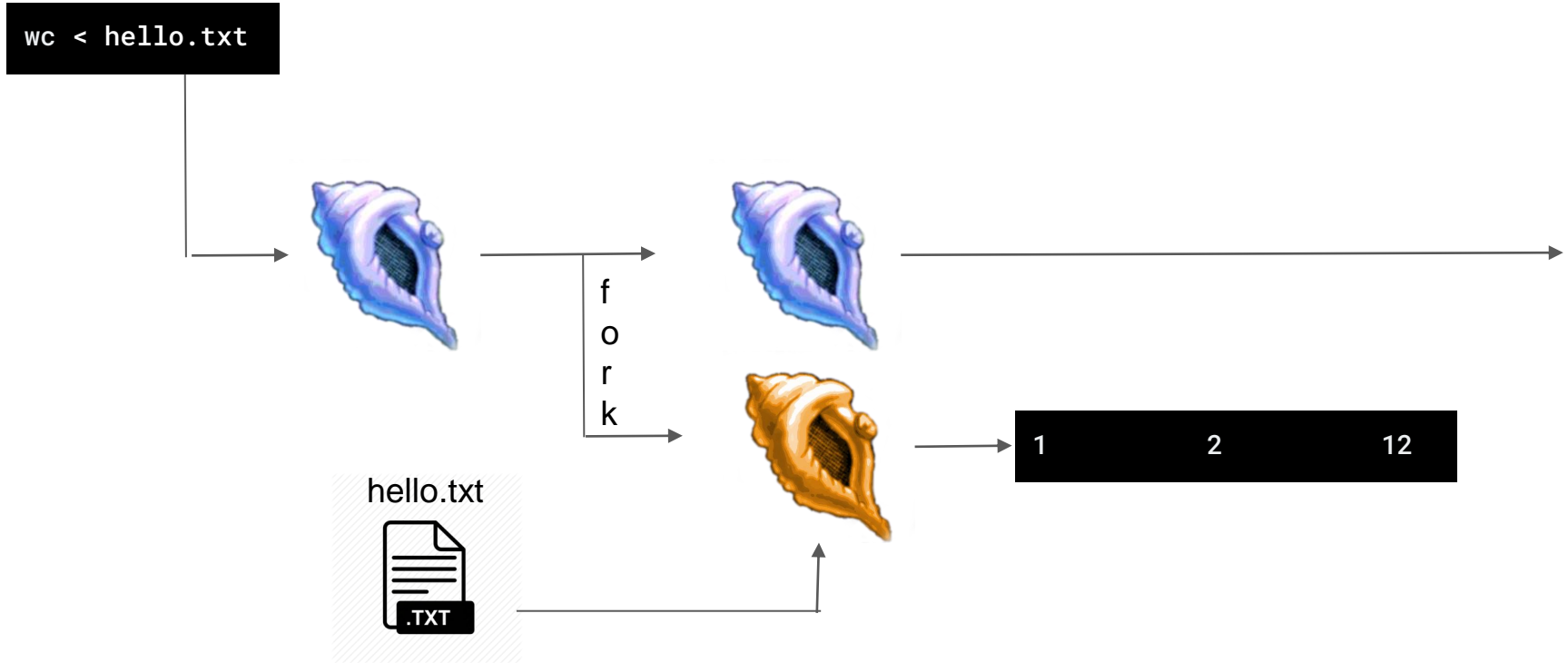
- > overwrites original file contents before writing out
- >> appends to the end of contents in file
- < read input from existing file rather than STDIN

I/O Redirection (Output)

```
echo 'Welcome to Systems!' > output.txt
```



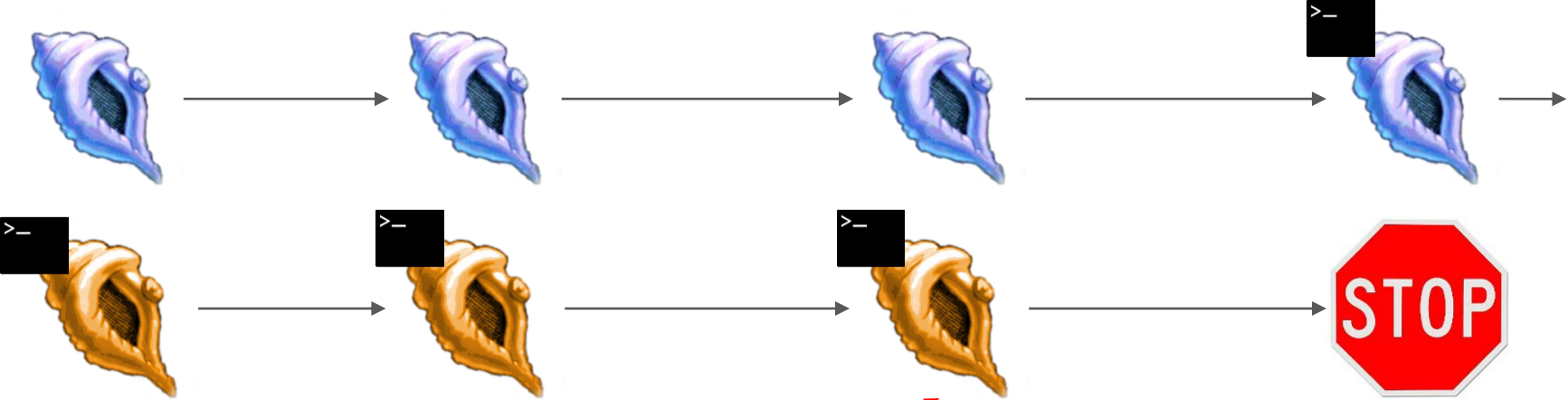
I/O Redirection (Input)



Signal Handling

- Shells can handle signals sent to them
 - SIGINT (Ctrl + C)
 - SIGTSTP (Ctrl + Z)
 - SIGCHLD (when a child process terminates)

Handling SIGINT (Ctrl + C)



1. Shell and single child process (in the foreground) are running

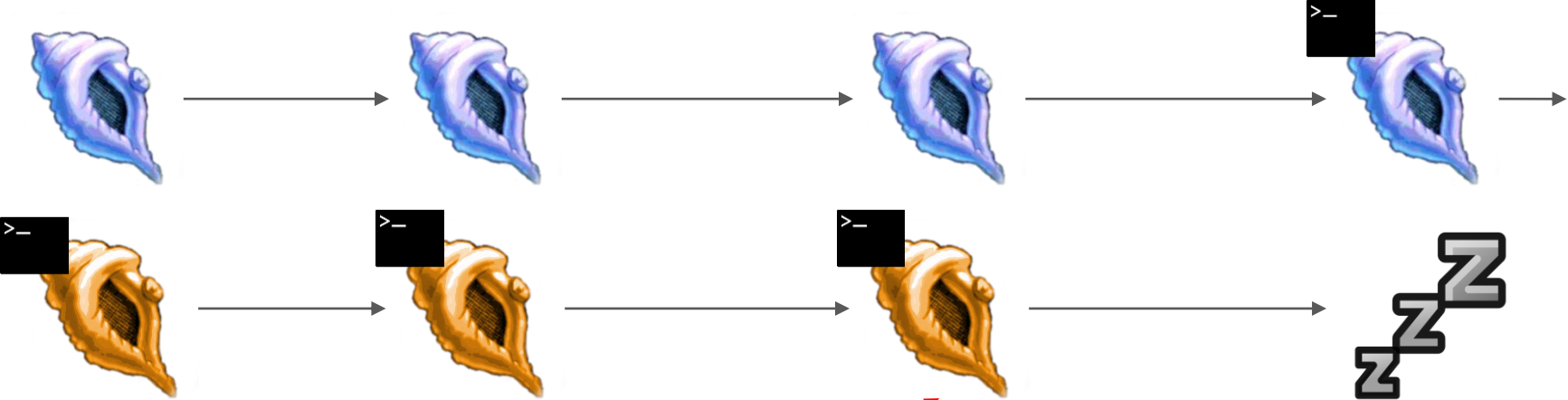
2. User sends SIGINT (Ctrl +C)

3. Signal sent to foreground process group

4. Group is forced to terminate, shell reacquires terminal control



Handling SIGTSTP (Ctrl + Z)



1. Shell and single child process (in the foreground) are running

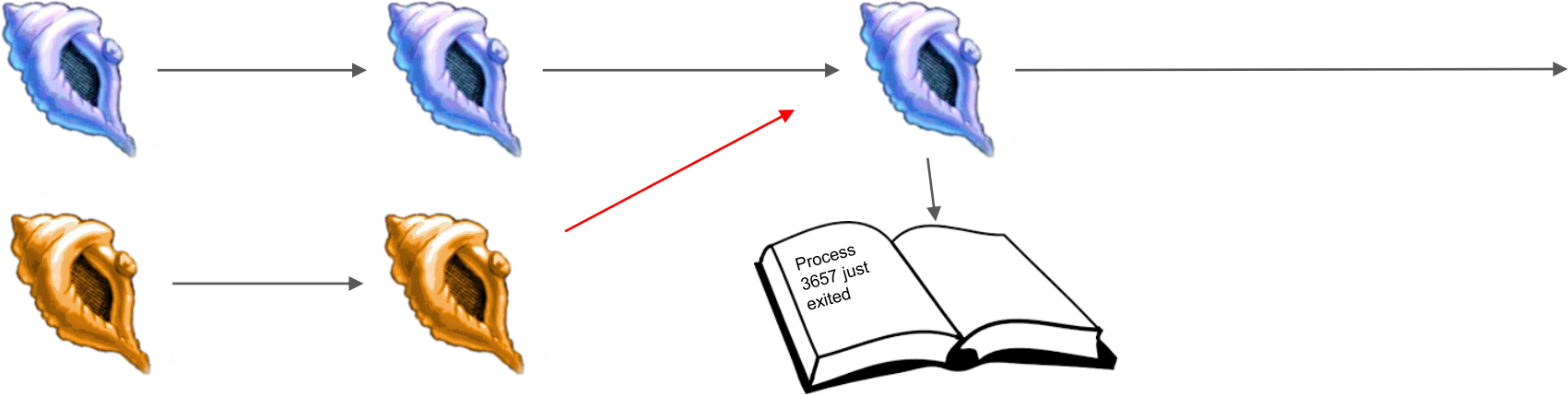
2. User sends SIGTSTP (Ctrl + Z)

3. Signal sent to foreground process group

4. Group is forced to stop, shell reacquires terminal control



Handling SIGCHLD



1. Shell and single child process (**foreground or background**) are running

2. Child process is finished and terminates - notifies parent by sending SIGCHLD

3. The shell's SIGCHLD handler code uses info to perform any necessary bookkeeping

4. Shell continues running

Handling SIGCHLD: WIF* Macros

- When `wait*` is called it will return a pid and a status for a child process that changes state. Using macros, we can decode this status to discover what state a process changed to and how it happened:
 - `WIFEXITED(status)` - did child process exit normally?
 - `WIFSIGNALED(status)` - was child process signaled to terminate?
 - `WIFSTOPPED(status)` - was child process signaled to stop?

Event	How to check for it	Additional info	Process stopped?	Process dead?
User stops fg process with Ctrl-Z	WIFSTOPPED	WSTOPSIG equals SIGTSTP	yes	no
User stops process with kill -STOP	WIFSTOPPED	WSTOPSIG equals SIGSTOP	yes	no
non-foreground process wants terminal access	WIFSTOPPED	WSTOPSIG equals SIGTTOU or SIGTTIN	yes	no
process exits via <code>exit()</code>	WIFEXITED	WEXITSTATUS has return code	no	yes
user terminates process with Ctrl-C	WIFSIGNALED	WTERMSIG equals SIGINT	no	yes
user terminates process with kill	WIFSIGNALED	WTERMSIG equals SIGTERM	no	yes
user terminates process with kill -9	WIFSIGNALED	WTERMSIG equals SIGKILL	no	yes
process has been terminated (general case)	WIFSIGNALED	WTERMSIG equals signal number	no	yes

Additional information can be found in the GNU C library manual, available at http://www.gnu.org/s/libc/manual/html_node/index.html. Read, in particular, the sections on Signal Handling and Job Control.

Additional Built-ins and extensions

- Your shell must contain two extra built-ins / functionality extensions
 - One high effort and one low effort (bolded is low-effort)
- Ideas include:
 - **Customizable Prompt**
 - **Setting/unsetting env vars**
 - Glob expansion (e.g., *.c)
 - Timing commands (ex. time)
 - Alias support
 - Embedded Apps
 - Shell Variables
 - Directory Stack
 - Command-line history
 - Backquote substitution
 - Smart command-line completion
- Unix Philosophy - implement only functionality that is not already supported using Unix commands. If you have an idea not shown on the list or have any doubts please ask us

Project Overview

Requirements and Grading

1. Basic Functionality - 50 pts
 - a. Start foreground and background jobs
 - b. Built-in commands : 'jobs', 'fg', 'bg', 'kill', 'stop'
 - c. Signal Handling (SIGINT, SIGTSTP, SIGCHLD)
 2. Advanced Functionality - 50 pts
 - a. I/O Pipes
 - b. I/O Redirection
 - c. Running programs requiring exclusive terminal access (ex: vim)
 3. Extra Built-ins - 20 pts
 4. Version Control (Git) - 10 pts
 5. Documentation - 10 pts
-

Total : 140 points

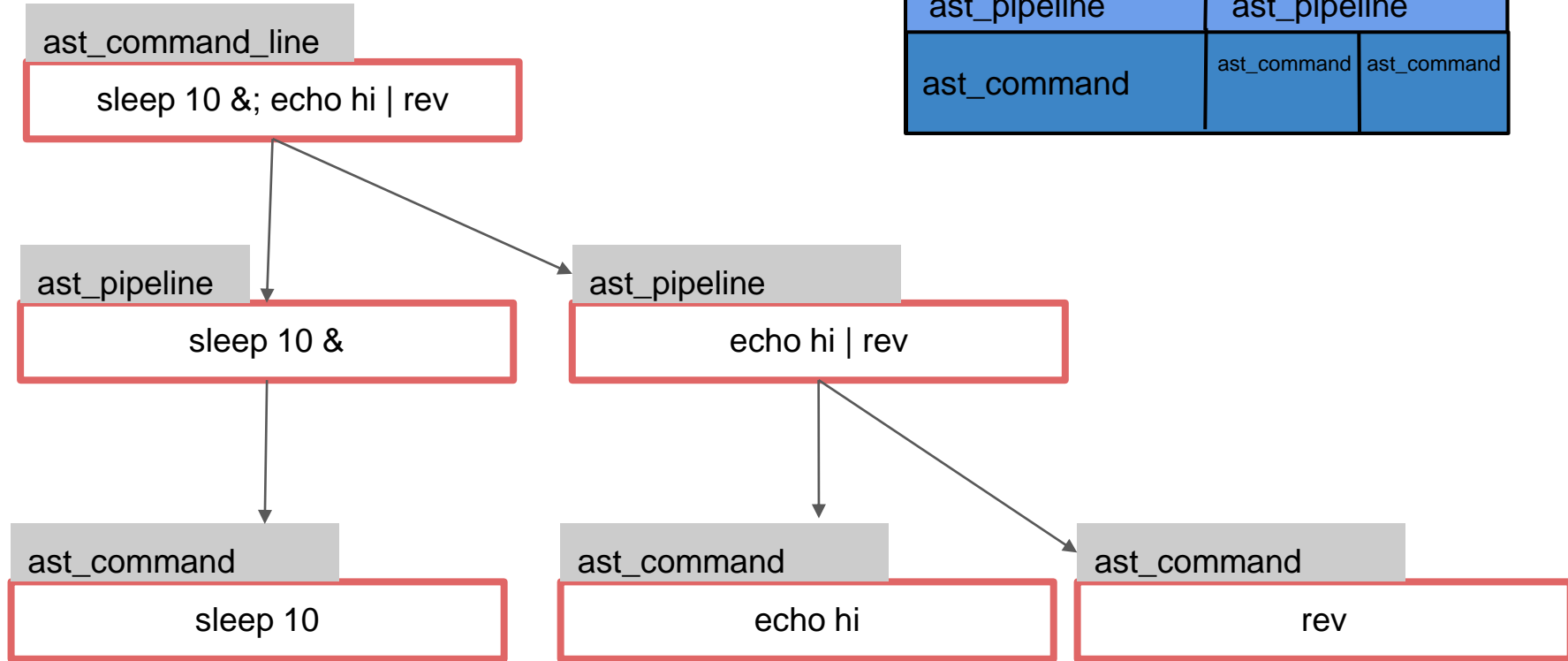
Before You Start Coding

- Take time to read over and comment the starter code
- Read the provided lecture material and Chapter 8 in the textbook
- Understand Exercise 1
 - `fork()` / `exec()` model
 - Piping : `pipe()`, `dup2()`, `close()`
- Check out Dr. Back's example shell
 - Located at `~cs3214/bin/cush-gback` in `rlogin`
 - Can be useful for comparing outputs with your shell

Base Code

- Already includes a parser!
- Parser spits out hierarchical data structures

Data Structures



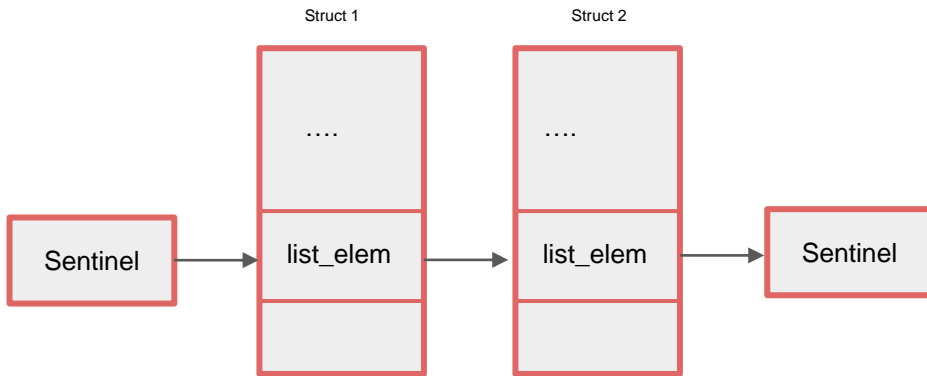
List Data Structure

- You're also provided with a linked list data structure
 - Check out list.h and list.c
- You'll be using this list throughout the semester
- Read through list.c before using it

“Data contains node” vs “Node points to data”

Your Linked List

```
struct list_elem {  
    struct list_elem * prev;  
    struct list_elem * next;  
}
```

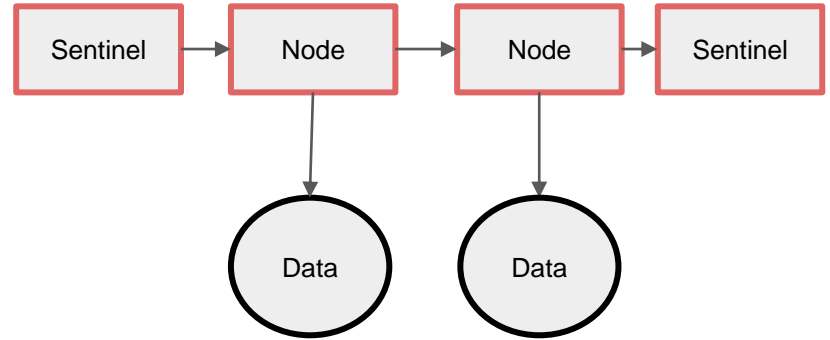


Retrieve data from a struct list_elem by using the **list_entry** macro:

```
struct ast_command * cmd = list_entry(e, struct ast_command, elem);
```


A Regular Linked List

```
class listnode<T> {  
    T data;  
    listnode<T> next;  
}
```



An example of an element in a list

```
struct ast_pipeline {
    struct list/* <ast_command> */ commands; /* List of commands */
    char *iored_input; /* If non-NULL, first command should read from
                        file 'iored_input' */
    char *iored_output; /* If non-NULL, last command should write to
                        file 'iored_output' */
    bool append_to_output; /* True if user typed >> to append */
    bool bg_job; /* True if user entered & */
    struct list_elem elem; /* Link element. */
};
```



Adding `list_elem` to a structure allows this structure to be added to a list

List Pitfalls

- Don't:

- Use the same list_elem for multiple lists
- Edit an element while iterating
 - Naive loop to remove elements in a list will fail!
- Forget to list_init()

BAD IDEA :(

```
// invalid example
for (list_elem in list)
{
    // do stuff

    if (someCondition)
    {
        list_remove(currElem);
    }
}
```

```
// valid example: deallocates a pipeline struct and any commands stored in it while iterating
void ast_pipeline_free(struct ast_pipeline *pipe)
{
    for (struct list_elem * e = list_begin(&pipe->commands); e != list_end(&pipe->commands); ) {
        struct ast_command *cmd = list_entry(e, struct ast_command, elem);
        e = list_remove(e); //Acts as the iterator; stores next element into e
        ast_command_free(cmd);
    }
    free(pipe);
} // make sure to remove an ast_pipeline from a list before adding it to another!
// bottom line with lists? ALWAYS TEST
```

Utility Functions (Strongly Recommended)

- Signal Support (signal_support.c / .h)
 - signal_block()
 - signal_unblock()
 - singal_set_handler()
- Terminal State Management (termstate_management.c / .h)
 - termstate_give_terminal_to()
 - termstate_give_terminal_back_to_shell()
 - termstate_restore()

Testing / Submission

- Please submit code that compiles!
- Test the driver before submitting, don't just run tests individually
- When grading, tests will be ran 3-5 times. If you crash a single time, it's considered failing

Test Driver

- The driver reads from .tst file that describes a test suite (ex. basic.tst)
 - Ex: basic.tst contains a series of test scripts that it will run from the folder /tests/basic

```
cd src/  
../tests/stdriver.py [options]
```

*- stdriver.py also available at ~cs3214/bin/stdriver.py

Options:

- -b : basic tests (processes, built-ins, signals)
- -a : advanced tests (I/O Piping, I/O Redirection, exclusive terminal access)
- -h : list all the options

Additional Tests

- You are required to write tests for your two extra built-ins
 - Create a .tst file in 'tests' and create a directory that will store your test scripts
- Inside <custom>.tst file:

```
= <custom> Tests  
pts <custom>/<test_name>.py  
pts <custom>/<test_name>.py  
...
```

```
= Milestone Tests  
1 basic/foreground.py  
1 basic/cmdfail_and_exit_test.py
```

- The driver checks number of total points (pts) to use for a test. Since this is just your own custom tests you can put an arbitrary number here

Additional Tests (Part 2)

- Make sure your custom.tst file is of type “ASCII text”

```
$ file custom.tst
```

```
custom.tst: ASCII text
```



- If it includes Windows terminators (CR, CRLF, etc.), see **man tr**
- We want `\n`, not `\r\n`

Design Document

- When you submit you must include a README.txt describing your implementation
- TAs will assign credit only for the functionality for which test cases and documentation exist

Submission. You must submit a design document, README.txt, as an ASCII document using the following format to describe your implementation:

```
Student Information
-----
<Student 1 Information>
<Student 2 Information>

How to execute the shell
-----
<describe how to execute from the command line>

Important Notes
-----
<Any important notes about your system>

Description of Base Functionality
-----
<describe your IMPLEMENTATION of the following commands:
jobs, fg, bg, kill, stop, ^C, ^Z >

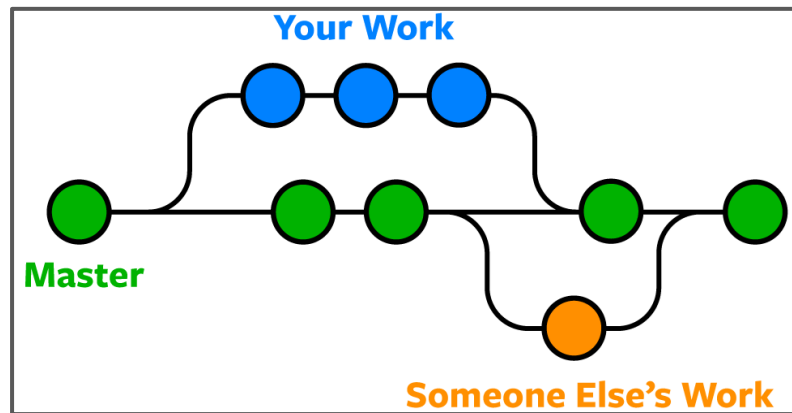
Description of Extended Functionality
-----
<describe your IMPLEMENTATION of the following functionality:
I/O, Pipes, Exclusive Access >

List of Additional Builtins Implemented
-----
(Written by Your Team)
  <builtin name>
  <description>
```

Version Control

Version Control

- You will be using Git for managing your source code
- Why?
 - Organizes your code
 - Keeps track of features
 - Allows collaborators to work freely without messing up other existing code
 - Back-ups whenever something goes wrong



Basic Git Commands

- Stage file for commit:

```
$ git add <file_name>
```

- Commit files:

```
$ git commit -m 'Add a description here'
```

- Push changes to remote (note: always pull before push!)

```
$ git push [origin <branch_name>]
```

Basic Git Commands

- Fetch changes from remote:

```
$ git pull
```

- Check status

```
$ git status
```

Setup Git Access

- You'll need an SSH Key to get access to projects at git.cs.vt.edu
- If you don't already have a key...

- Create a new key:

```
$ ssh-keygen -t rsa -b 4096 -C "email@vt.edu" \  
-f ~/.ssh/id_rsa
```

- Add Key to <https://git.cs.vt.edu/profile/keys>
 - You will paste public key here ----->

Add an SSH key

To add an SSH key you need to [generate one](#) or use an [existing key](#).

Key

Paste your public SSH key, which is usually contained in the file '~/.ssh/id_ed25519.pub' or '~/.ssh/id_rsa.pub' and begins with 'ssh-ed25519' or 'ssh-rsa'. Don't use your private SSH key.

Typically starts with "ssh-ed25519 ..." or "ssh-rsa ..."

Title

Name your individual key via a title

Verify Git Access

- Verify you have access
- The first time you connect you will be asked to verify the host, just answer 'Yes' to continue

```
11 spencetk@linden ~ >ssh git@git.cs.vt.edu
```

```
PTY allocation request failed on channel 0
```

```
Welcome to GitLab, @spencetk! ← Your pid should be displayed here
```

```
Connection to git.cs.vt.edu closed.
```

- You can get in-depth explanations here:
 - [Generate a key](#)
 - [Use an existing key](#)

GitLab Project Setup

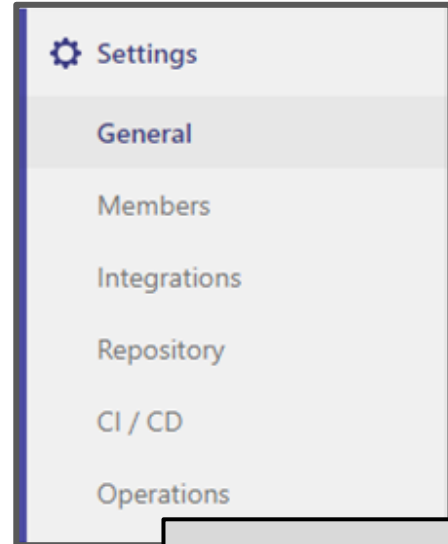
1. One member will fork the base repository:
 - <https://git.cs.vt.edu/cs3214-staff/cs3214-cush>
2. Invite partner to collaborate
 - Go to Settings > Members to add them
 - Check partner role permissions too
3. Both members will clone the forked repository on their machines:

```
$ git clone <your git repo url>.git
```

```
cs3214-cush
```

IMPORTANT: Set forked repository to **private**

Go to Settings > General > Visibility, project features, permissions



*Your forked repository will have a navigation menu on the left side. Click under Settings to add members and set repo to private

The GNU Project Debugger

Starting GDB

- Invoke GDB with a program and arguments:

```
$ gdb --args program arg1 arg2
```

- Better alternative:

```
(gdb) run arg1 arg2
```

- Must be compiled with debug symbols, -g

Breakpoints

- Set a breakpoint

```
(gdb) b <func_name> OR  
(gdb) b <line_number>
```

- Set a conditional breakpoint:

```
(gdb) b <func_name> if <condition>
```

- Ignore breakpoint #1 100 times

```
(gdb) ignore 1 100
```

- Show # of times breakpoint was hit

```
(gdb) info b
```

Backtrace and Frames

- Show backtrace:

```
(gdb) backtrace
```

- Show frame:

- After selecting frame, you can print all variables declared in that function call

```
(gdb) frame <num>
```

Follow-Fork-Mode

- Which process to follow after a fork (parent / child):

```
(gdb) set follow-fork-mode <mode>
```

- 'parent' = ignore child process and continue debugging the parent
 - 'child' = begin debugging the child process when fork() is called
- Retaining debugger control after fork:
 - After a fork, specify whether to freeze the child or allow it to run (this may make it difficult to find race conditions)

```
(gdb) set detach-on-fork <mode>
```

Layout Source

- Show source code lines while debugging
- Far superior alternative to 'list'
- Toggle with Ctrl-X+A

```
(gdb) layout src
```

Advice

How Can I Fail Systems?

- Not utilize class resources
- Mismatch your time
- Not understand your tools
- Fail to get along with your partner
- Not break down the problem
- Not understand the concepts

Advice

- START EARLY
- Create a roadmap before starting projects
- Utilize TAs
 - Come with questions prepared, try to figure out what the problem is first
 - **Be organized and have clean code** - the cleaner it is, the faster we can help!
 - **Run valgrind and try debugging with GDB before consulting us**
 - Discord, Zoom, Class Forum
- Understand the Exercises
- Use valgrind! This can isolate many bugs
- Become an expert at the debugger
- Find what works best for communicating with your partner
 - Discord, Zoom, etc.

Sources

- Referred to previous help session slides created by previous UTA's Kent McDonough, Connor Shugg, Joe D'Anna, Chris Cerne, and Justin Vita for since the Spring 2021 Semester
- Spencer Keefer created the revised slides

Thanks for attending!
Questions?