CS 3214: Project 1

The Customizable Shell

Help Session:
Friday Feb 9, 2023 7:00 PM

Anthony Nguyen <anthonyn33@vt.edu>
Thomas Tran <thomastran@vt.edu>

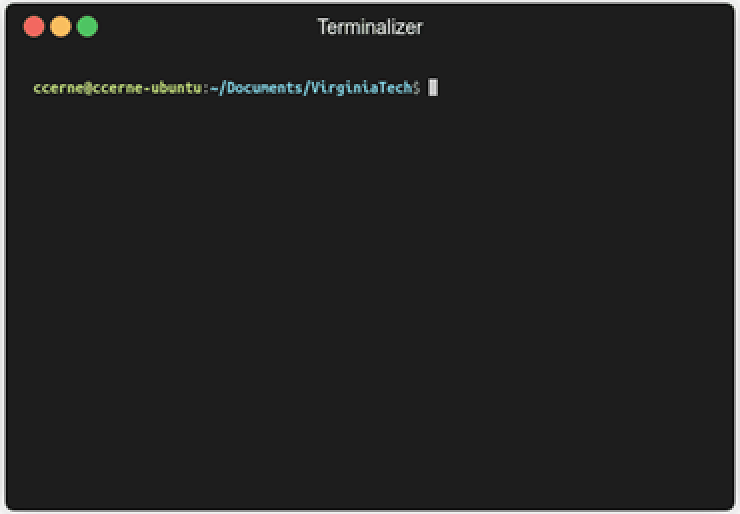
Topics

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

Shell Concepts

What is a shell?

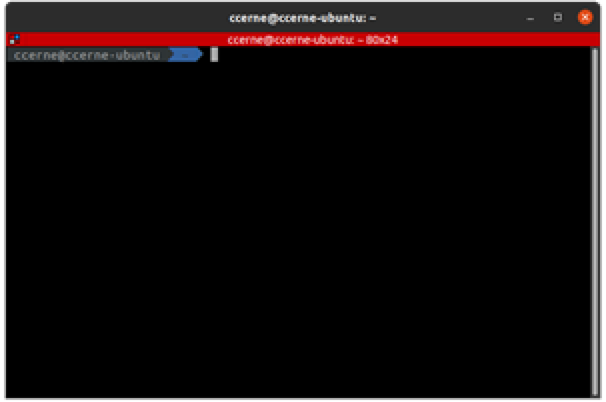
- Command Interpreter
    - Reads user input and executes user requests
    - Not to be confused with a “Terminal” (next slide explains distinction)



```
ccerne@ccerne-ubuntu:~/Documents/VirginiaTech$
```

Terminal vs Shell

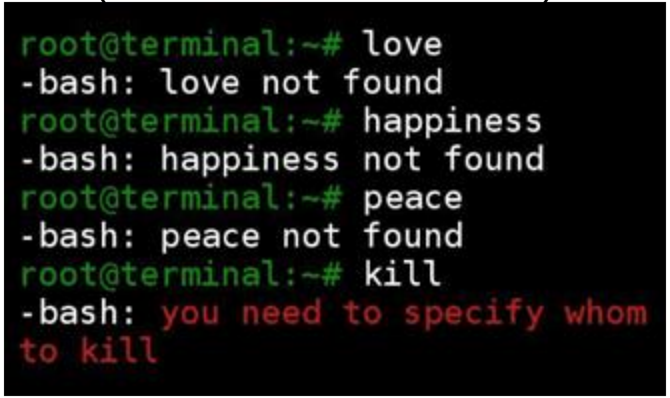
Terminal (the front-end GUI of our shell)



```
ccerne@ccerne-ubuntu  ~
```

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

Shell (an executable with no GUI)



```
root@terminal:~# love
-bash: love not found
root@terminal:~# happiness
-bash: happiness not found
root@terminal:~# peace
-bash: peace not found
root@terminal:~# kill
-bash: you need to specify whom
to kill
```

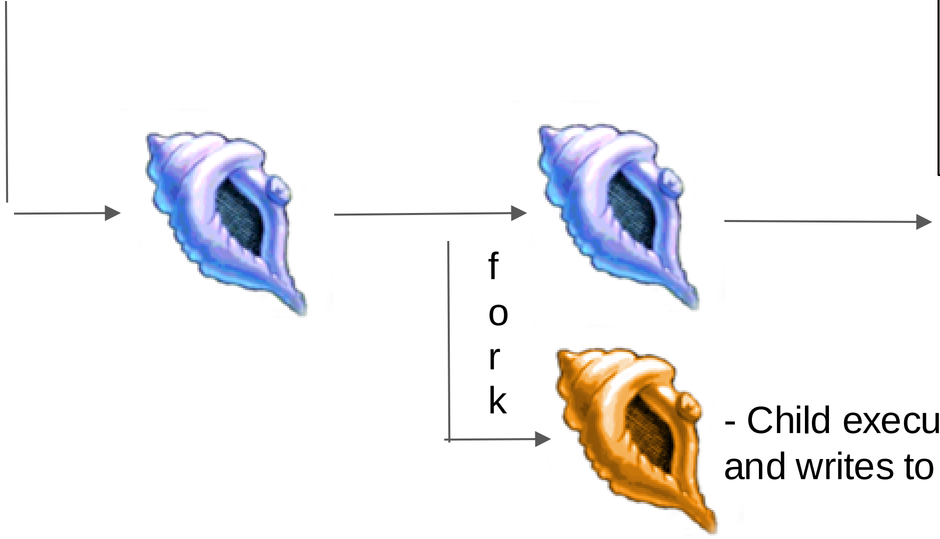
This terminal is running bash, a shell program

Behind the Scenes

FOUR STEPS for *non-built-in*

1. Shell waits for user input
2. Shell interprets command
3. Forks a process
4. If the command is a foreground process, the parent waits for the child to finish before taking in new command. If it's a background process, the parent waits for a signal from the child but at the meantime accepts additional command.

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



```
Welcome to Systems!
```

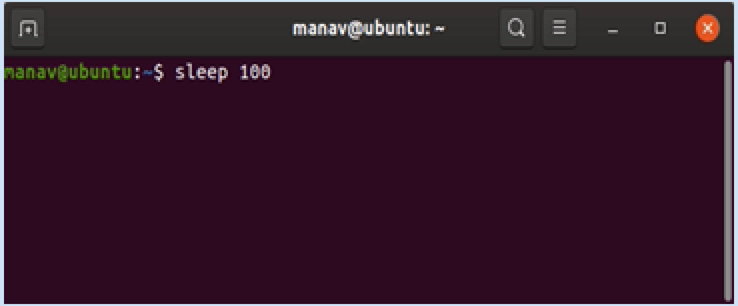
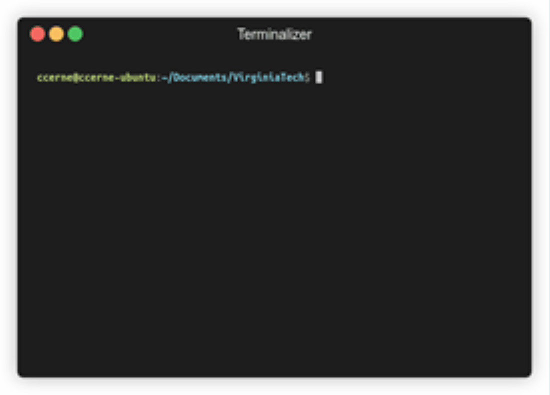
- Child executes the command and writes to stdout

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
- Terminal State

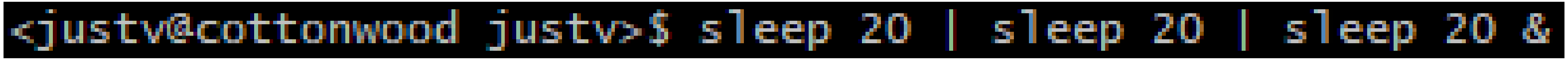
Foreground / Background Processes

- The shell can fork processes into the foreground or background

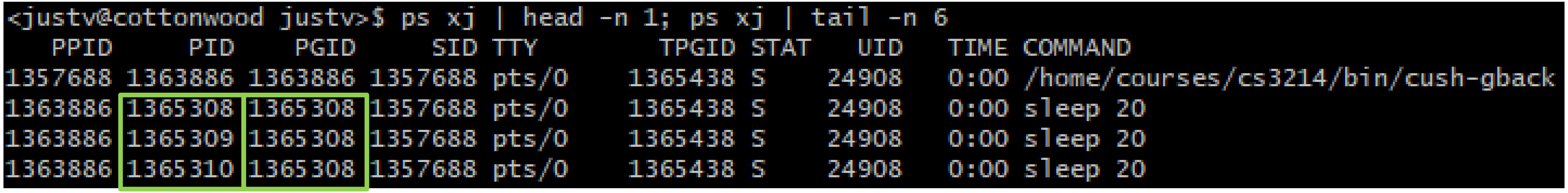
Foreground	Background
- Only one foreground process group at a time - Process has access to the terminal  manav@ubuntu: ~ `manav@ubuntu:~$ sleep 100`	- Process doesn't have terminal access - Using '&' sends command to background to run  Terminalizer `ccerne@ccerne-ubuntu:~/Documents/VirginiaTech$`

Process Groups

- A Job is essentially a pipelined-command
- Each Job has its own process group
  - Each command within a Job should have the same PGID
  - Two methodologies of creating new processes:
    - fork() and execvp()
    - 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!

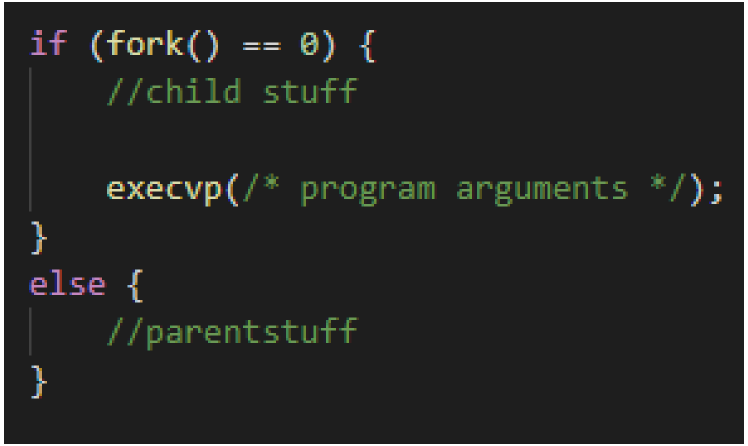
POSIX Spawn

- Replaces fork() + exec() entirely
- Code is streamlined rather than handling multiple processes in if-else statements
- posix_spawnattr_t and posix_spawn_file_actions_t are structs that store information about process groups and I/O redirection/piping respectively. These structs don’t do anything until posix_spawnp is used.
    - You will need to setup/configure these structs

- Example: posix_spawn(3) - Linux manual page (man7.org)

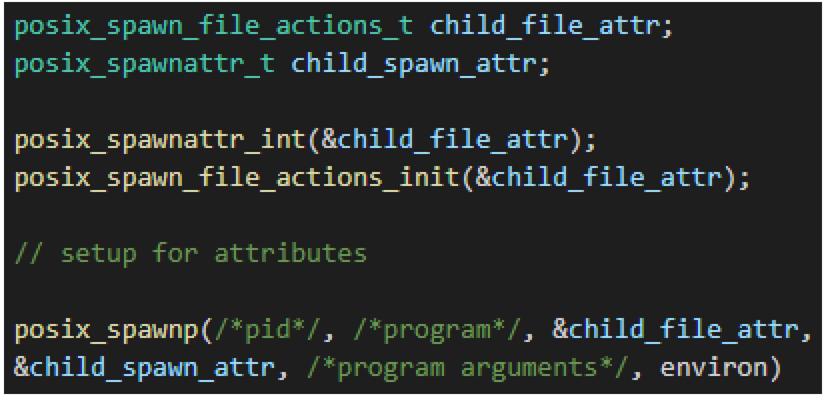
Note: You need to include “spawn.h” in your cush.c to use these functions. The file is located in the posix_spawn directory. Also be sure to use the “make” command to compile posix_spawn.

fork() + exec()



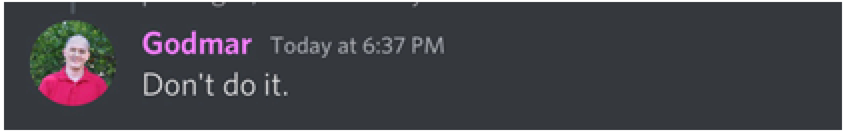
```
if (fork() == 0) {
    //child stuff

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

You can use fork() + exec() for this project, but our recommendation is:



Godmar Today at 6:37 PM

Don't do it.

POSIX Spawn Attributes

- Process Groups - posix_spawnattr_setpgroup()
- Terminal Control - posix_spawnattr_tcsetpgrp_np()
- Piping - posix_spawn_file_actions_adddup2()
- I/O Redirection - posix_spawn_file_actions_addopen()

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

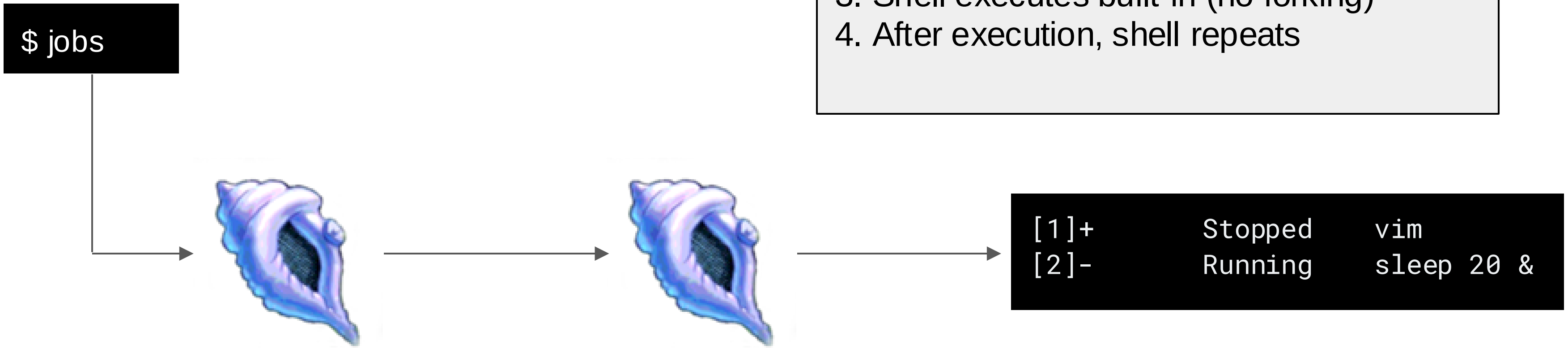
Built-in Commands

- 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
  - exit - exits the shell
- Built-in Commands are not considered Jobs
- Two additional built-ins / functionality extenders also required (examples in later slide)
  - One low-effort (cd, custom prompt, etc.)
  - One high-effort (glob expansion, history, etc.)

Built-ins Behind the Scenes

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

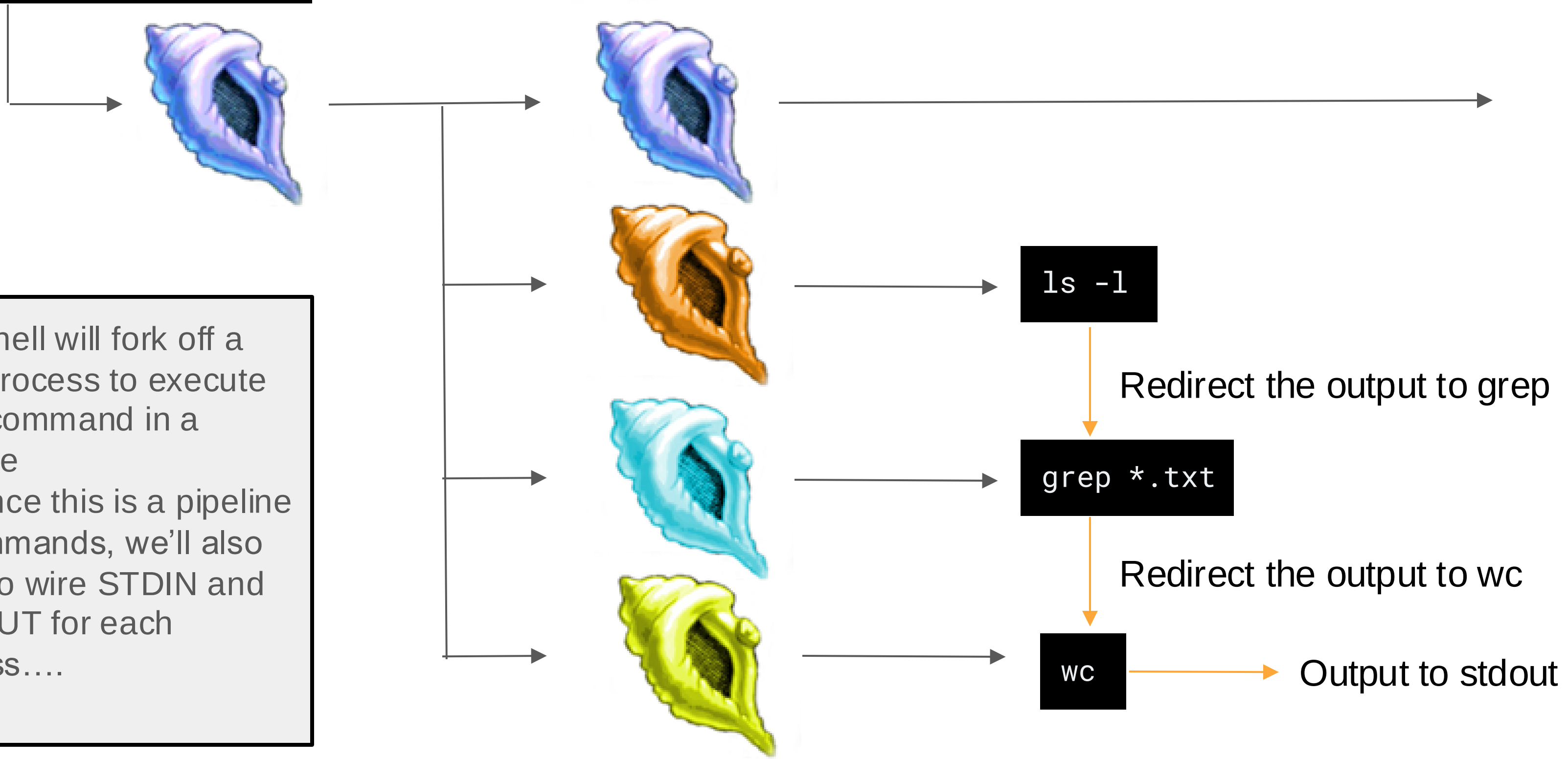


```
$ jobs
```

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

I/O Piping

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



`ls -l`

Redirect the output to grep

`grep *.txt`

Redirect the output to wc

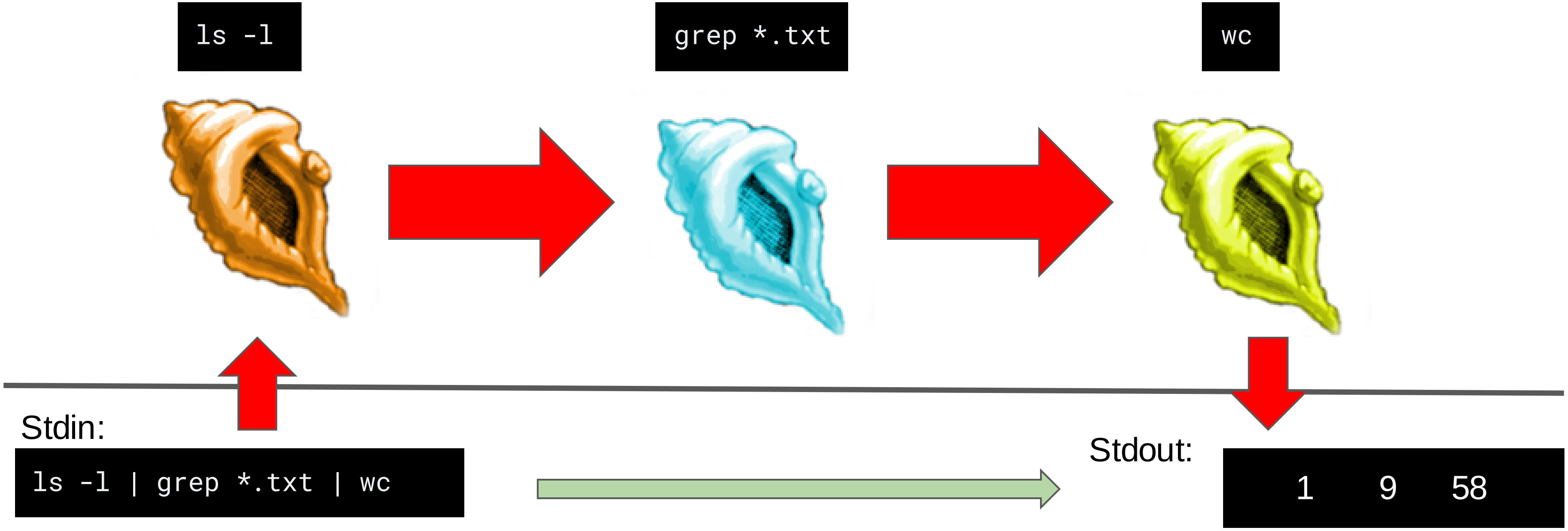
`wc`

Output to stdout

- 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



```
ls -l
```

```
grep *.txt
```

```
wc
```

Stdin:

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

Stdout:

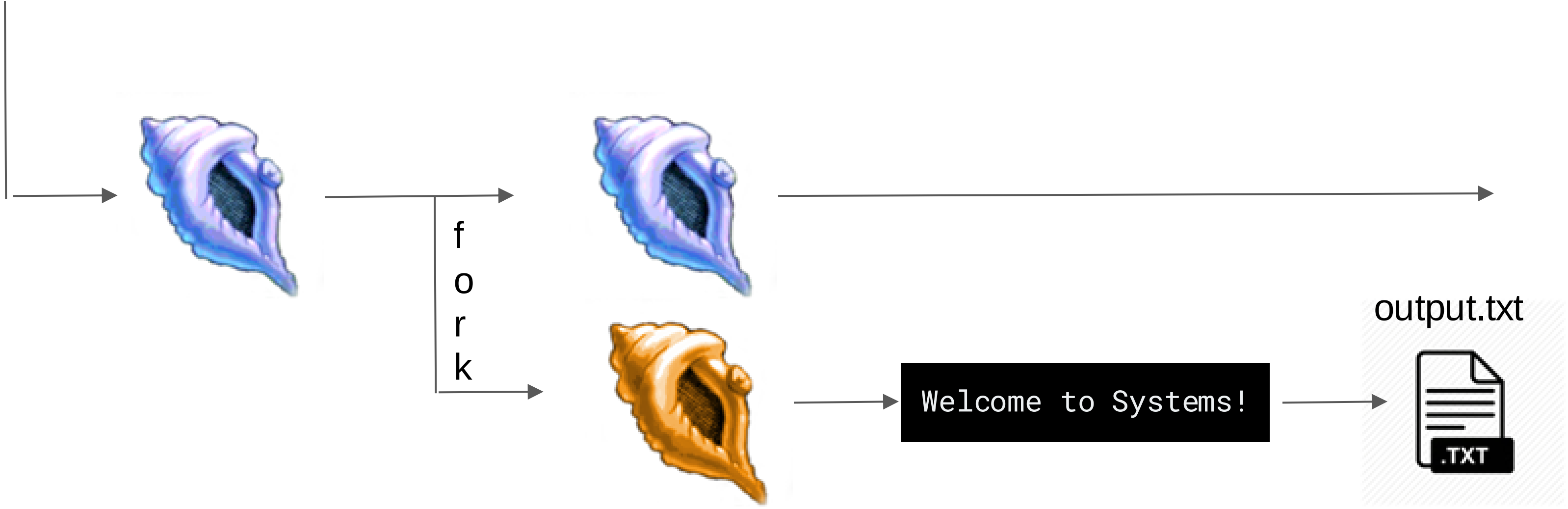
```
1    9    58
```

I/O Redirection

- > overwrites original file contents before writingnew output
- >> appends new content to the end of the original file
- < read input from a file rather than STDIN

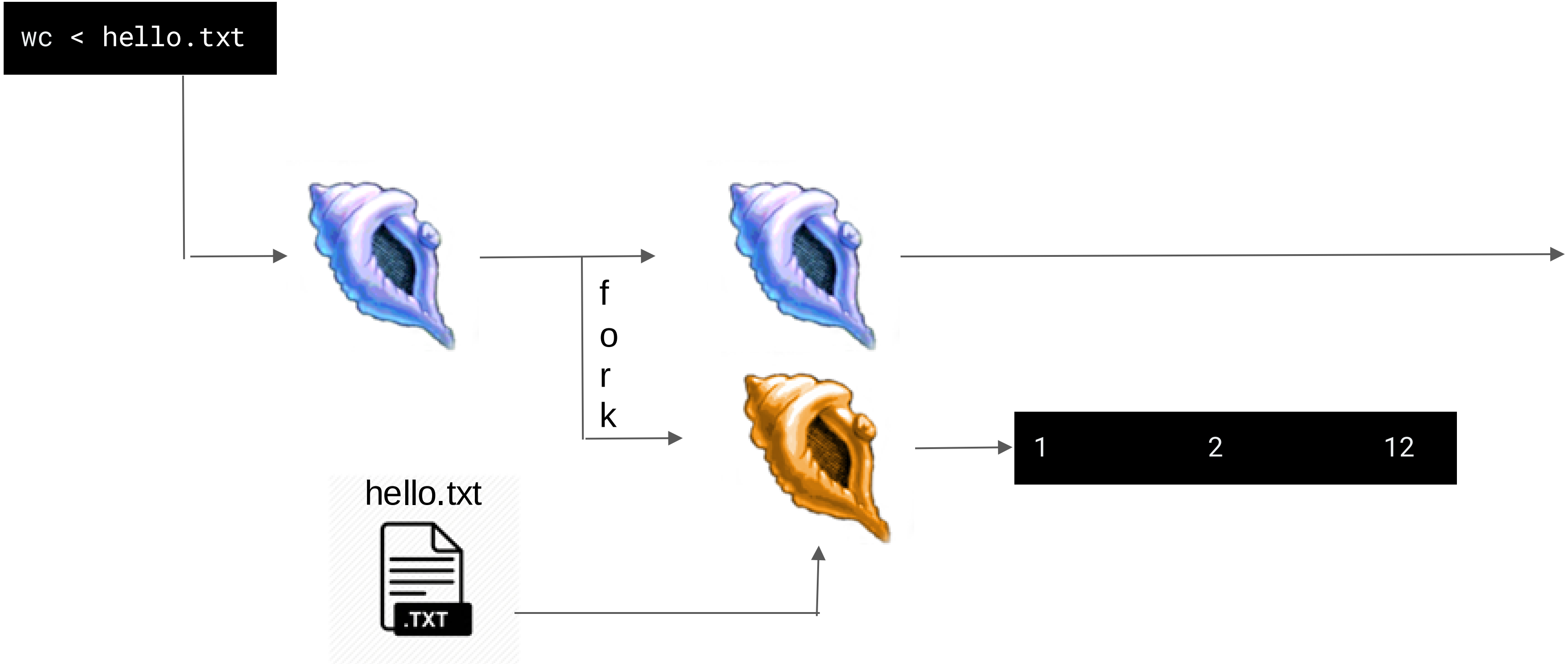
I/O Redirection (Output)

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



```
Welcome to Systems!
```

I/O Redirection (Input)

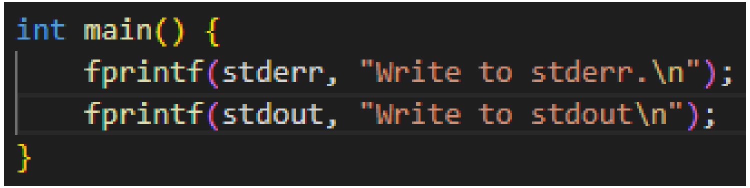


```
wc < hello.txt
```

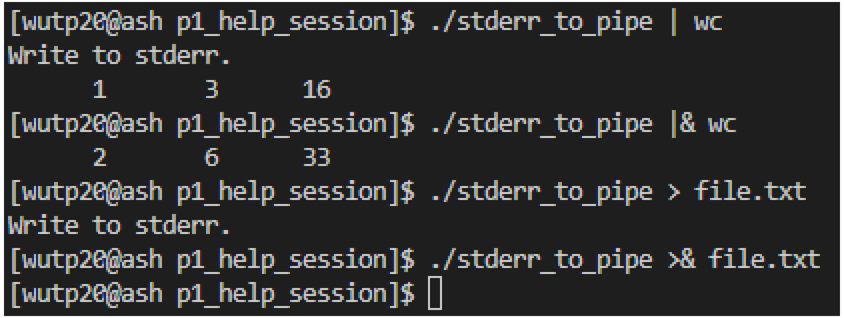
```
1          2          12
```

I/O Redirection (Stderr)

- Contents written to STDERR can also be piped into other processes using |& and outputted to files using >&.



```c
int main() {
    fprintf(stderr, "Write to stderr.\n");
    fprintf(stdout, "Write to stdout\n");
}
```



```
[wutp20@ash p1_help_session]$ ./stderr_to_pipe | wc
Write to stderr.
      1       3      16
[wutp20@ash p1_help_session]$ ./stderr_to_pipe |& wc
      2       6      33
[wutp20@ash p1_help_session]$ ./stderr_to_pipe > file.txt
Write to stderr.
[wutp20@ash p1_help_session]$ ./stderr_to_pipe >& file.txt
[wutp20@ash p1_help_session]$ 
```

Notice how the message “Write to stderr.” was not outputted.

Signal Handling

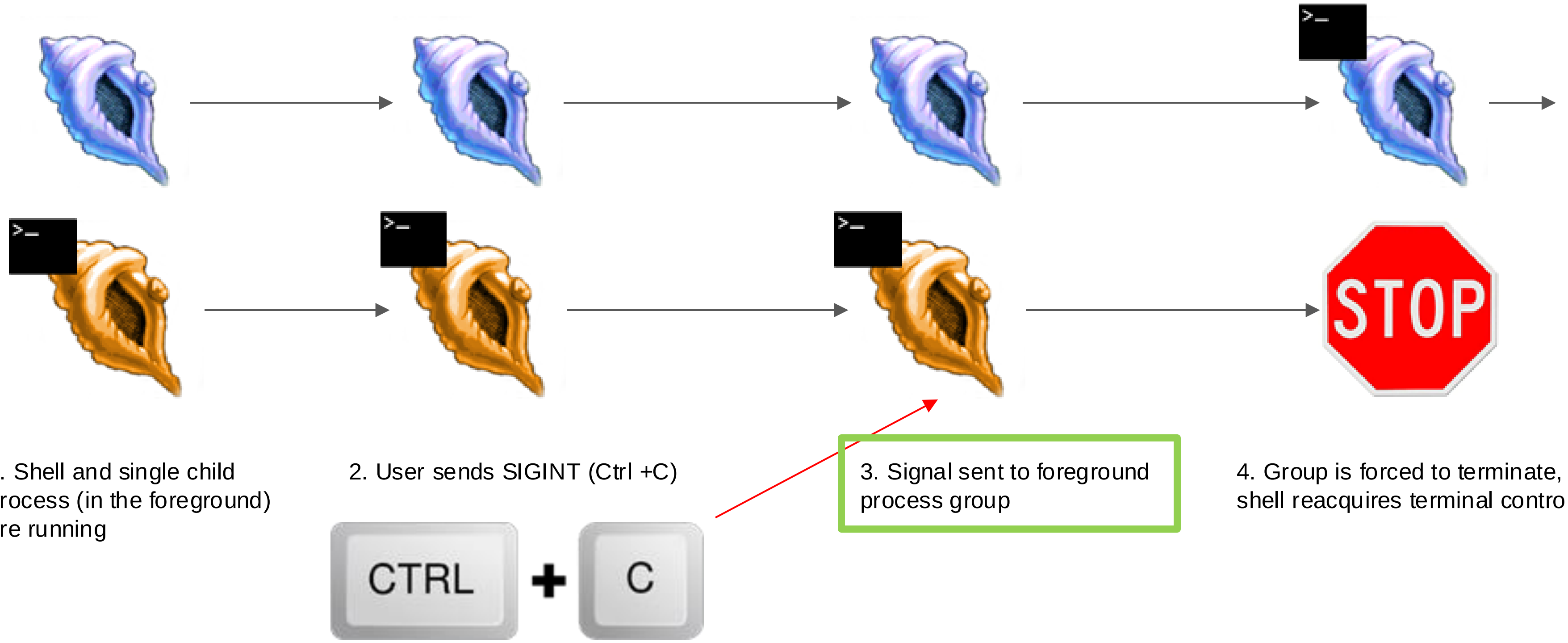
- Shells can handle signals sent to them
    - SIGINT (Ctrl + C)
    - SIGTSTP (Ctrl + Z)
    - SIGCHLD (when a child process terminates)
        - Cannot predict when SIGCHLD arrives, make sure to follow async-signal safety rules (refer to slide 9 of L-P6)

- Most of the functionality of this will be done in handle_child_status(pid_t pid, int status)

- User processes can use the kill(2)/killpg(2) system call to send signals to each other (subject to permission)

- strsignal(3) returns a string describing the signal number passed in

Handling SIGINT (Ctrl + C)



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

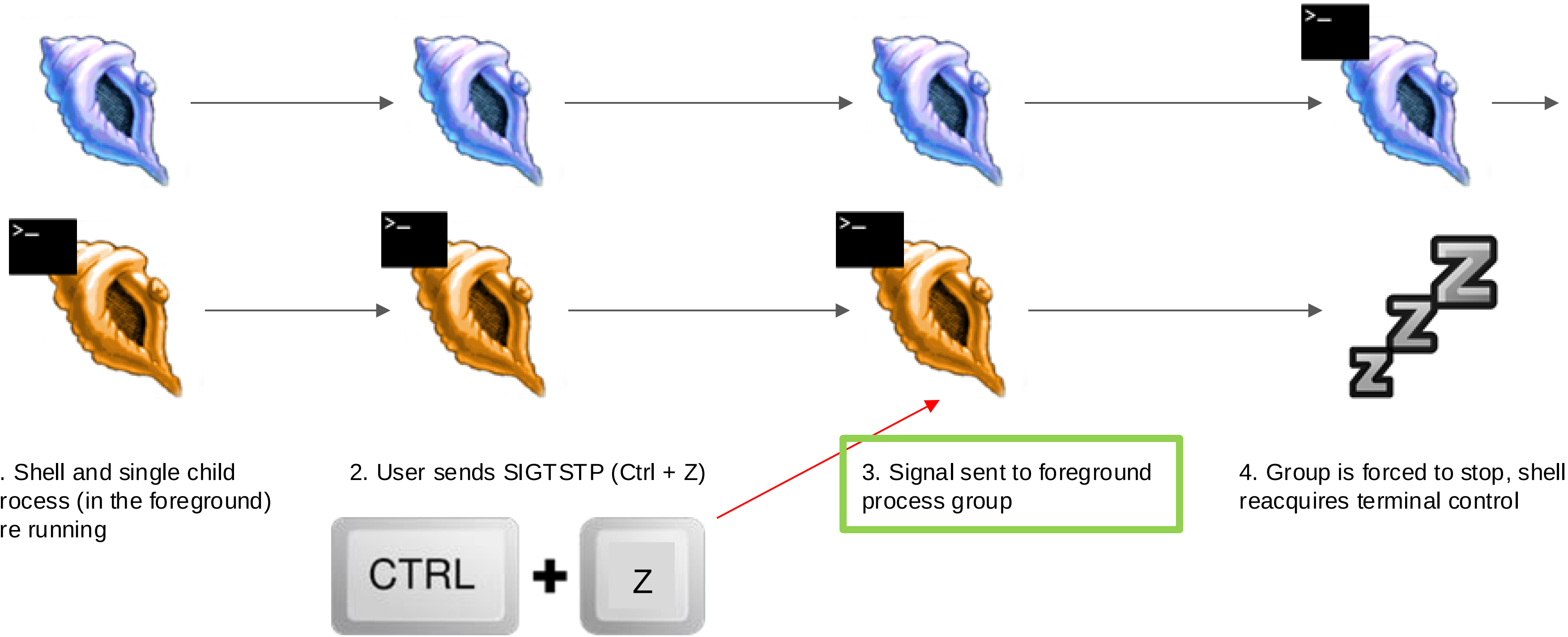
2. User sends SIGINT (Ctrl +C)

3. Signal sent to foreground process group

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

Explanation : https://en.wikipedia.org/wiki/Process_group#Applications

Handling SIGTSTP (Ctrl + Z)



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

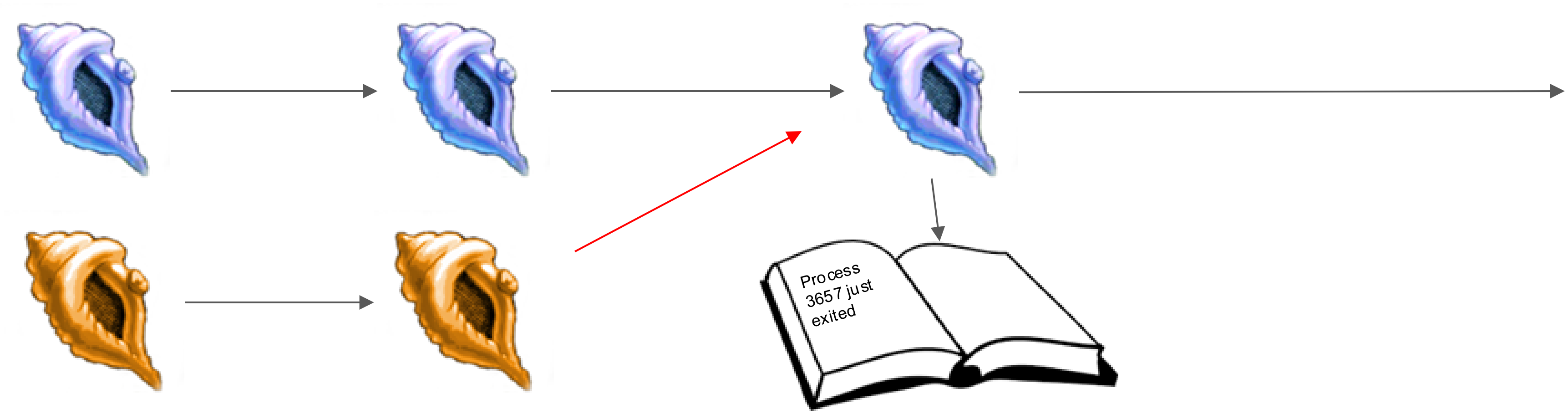
2. User sends SIGTSTP (Ctrl + Z)

3. Signal sent to foreground process group

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

Explanation : https://en.wikipedia.org/wiki/Process_group#Applications

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 to capture child status and perform any necessary bookkeeping (in next slide)

4. Shell continues running

Explanation : https://en.wikipedia.org/wiki/Process_group#Applications

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 `exit()`	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.

Terminal State

- Sample: make the current terminal state the new known “good state” of your shell
  - `void termstate_sample(void)`
  - Sample when a (originally started, currently) foreground job exits normally with status 0
- Save: save the current terminal state for a specific job (e.g. when its stopped) so it can be restored when the job moves back to the foreground
  - `void termstate_save(struct termios *saved_tty_state)`

Project Overview

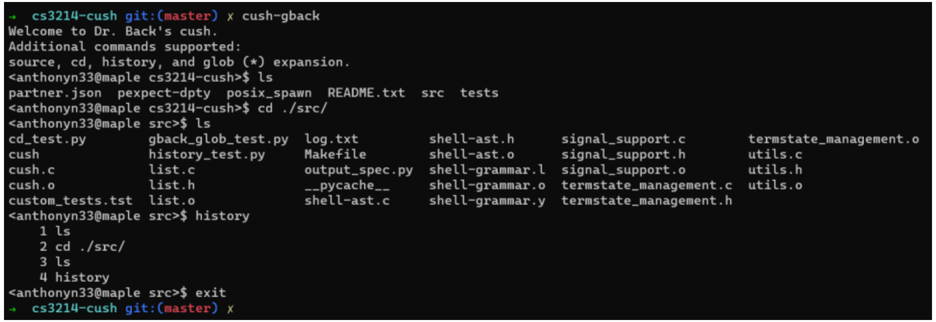
Requirements and Grading

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 Piping
    b. I/O Redirection
    c. Running programs requiring exclusive terminal access (ex: vim)
3. Two Extra Built-ins - 20 pts
    a. One low effort
    b. One high effort
    c. Testing required for both
4. Version Control (git) - 10 pts
    a. At least 3 commits per partner
5. Documentation - 10 pts
    a. Write a README.txt
    b. Comments and function headers

Total : 140 points

Before You Start Coding ….

- Take time to read over, understand the spec and the starter code
- Read the provided lecture material
- Understand Exercise 1
    - fork() / exec() model (please just read posix_spawn)
    - Piping : pipe(), dup2(), close()
- Check out Dr. Back's example shell
    - **~cs3214/bin/cush-gback** in rlogin
    - Compare its output with your shell's



```
→  cs3214-cush git:(master) ✗ cush-gback
Welcome to Dr. Back's cush.
Additional commands supported:
source, cd, history, and glob (*) expansion.
<anthonyn33@maple cs3214-cush>$ ls
partner.json  pexpect-dpty  posix_spawn  README.txt  src  tests
<anthonyn33@maple cs3214-cush>$ cd ./src/
<anthonyn33@maple src>$ ls
cd_test.py        gback_glob_test.py  log.txt         shell-ast.h      signal_support.c        termstate_management.o
cush              history_test.py     Makefile        shell-ast.o      signal_support.h        utils.c
cush.c            list.c              output_spec.py  shell-grammar.l  signal_support.o        utils.h
cush.o            list.h              __pycache__     shell-grammar.o  termstate_management.c  utils.o
custom_tests.tst  list.o              shell-ast.c     shell-grammar.y  termstate_management.h
<anthonyn33@maple src>$ history
    1 ls
    2 cd ./src/
    3 ls
    4 history
<anthonyn33@maple src>$ exit
→  cs3214-cush git:(master) ✗
```

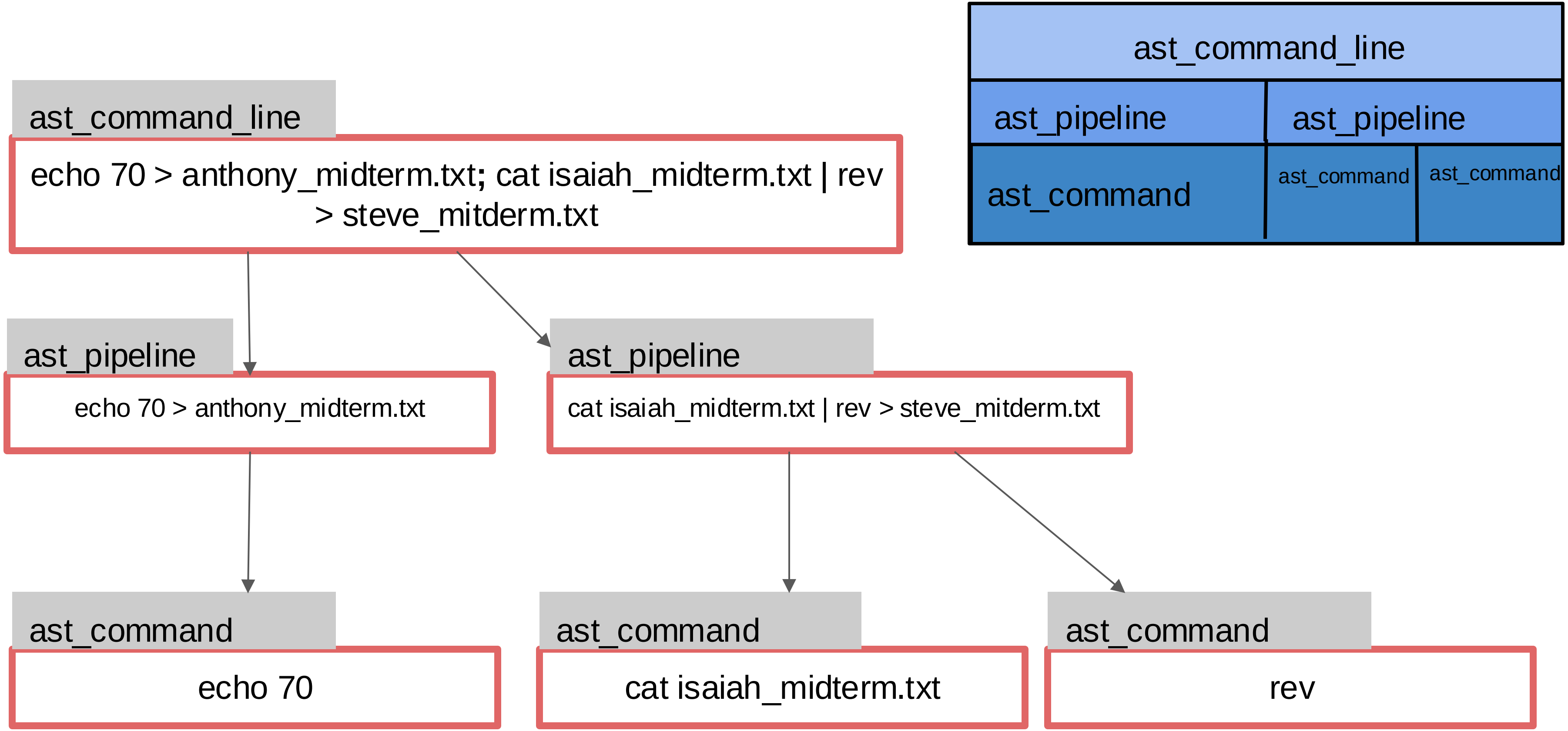
Base Code

- Already includes a parser!
- Parser spits out hierarchical data structures
- All you need to do is process these structs!

Structs

- ast_command_line
- ast_pipeline
  - A “Job”
  - I/O redirection files
  - Append (>>)
  - Background (&)
- ast_command
  - Argv
  - Stderr redirect

See **shell-ast.h** for fields and their descriptions!



ast_command_line

echo 70 > anthony_midterm.txt**;** cat isaiah_midterm.txt | rev > steve_mitderm.txt

ast_pipeline

echo 70 > anthony_midterm.txt

ast_pipeline

cat isaiah_midterm.txt | rev > steve_mitderm.txt

ast_command

echo 70

ast_command

cat isaiah_midterm.txt

ast_command

rev

ast_command_line

ast_pipeline

ast_pipeline

ast_command

ast_command

ast_command

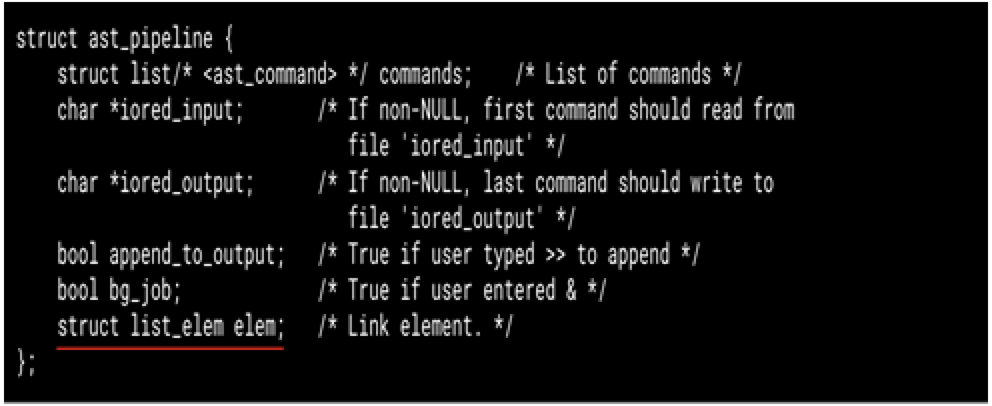
List Data Structure

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

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

Our Linked List

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



```
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. */
};
```

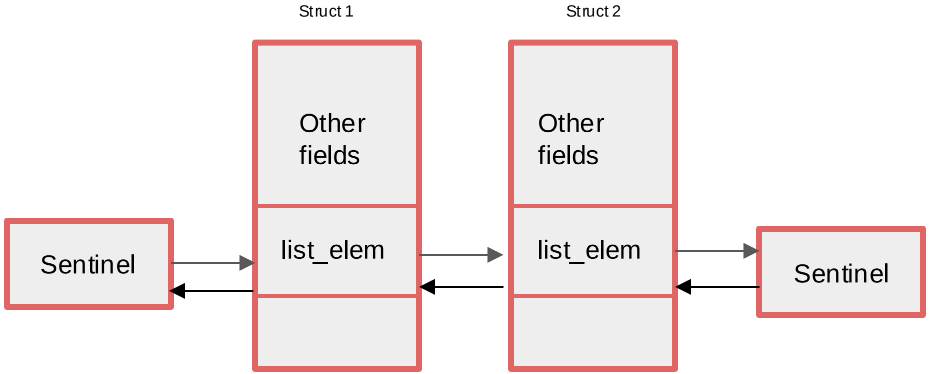
A Regular Linked List

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

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

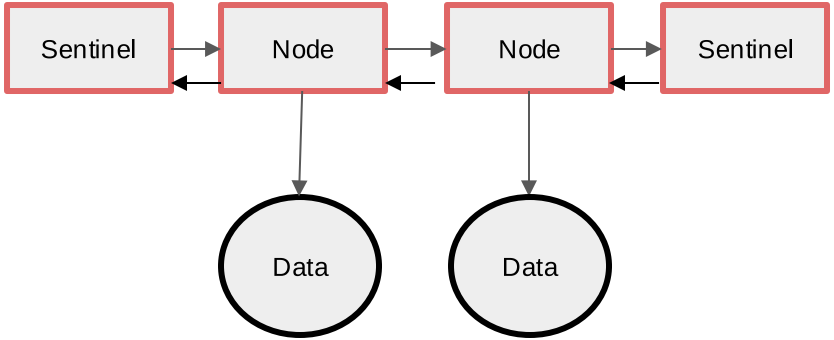
Our Linked List

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



A Regular Linked List

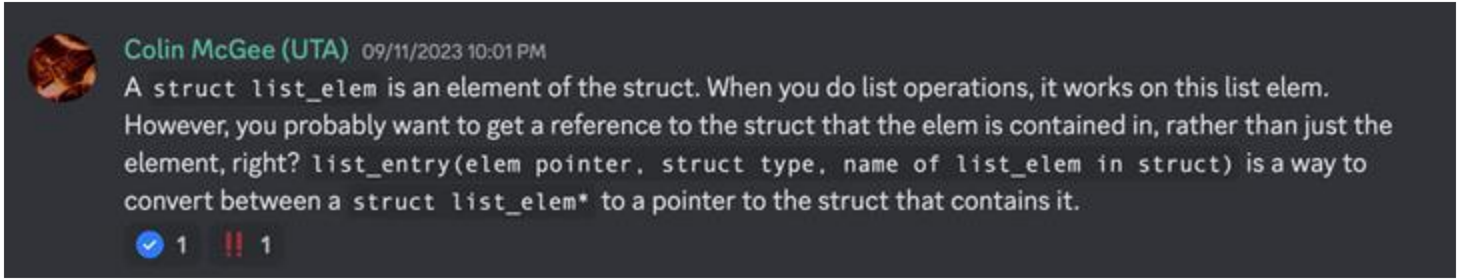
```
class listnode<T> {
                T data;
                listnode <T> prev;
                listnode<T> next;
}
```



So how do I get my data?

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

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



Colin McGee (UTA) 09/11/2023 10:01 PM

A `struct list_elem` is an element of the struct. When you do list operations, it works on this list elem. However, you probably want to get a reference to the struct that the elem is contained in, rather than just the element, right? `list_entry(elem pointer, struct type, name of list_elem in struct)` is a way to convert between a `struct list_elem*` to a pointer to the struct that contains it.

✓ 1 ‼ 1

Beautiful explanation by one of our UTA’s :)

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_init()
  - termstate_give_terminal_to()
  - termstate_give_terminal_back_to_shell()
  - termstate_get_current_terminal_owner()
  - termstate_save()
  - termstate_restore()

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***
  - ***Implementing the ‘cd’ built-in***
  - Glob expansion (e.g., *.c)
  - Timing commands (ex. time)
  - Alias support
  - Shell Variables
  - Directory Stack
  - Command-line history
  - Backquote substitution
  - Smart command-line completion
  - Embedded Apps
- **If you have an idea not shown on the list or have any doubts please ask us**

Testing / Submission

- 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
- Make sure you don’t have undefined behavior by checking the system call return code and using valgrind to address memory related issues

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
…
```

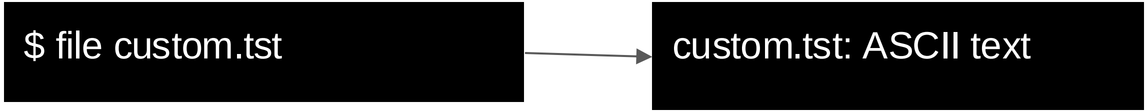
```
= Milestone Tests
1 basic/foreground.py
1 basic/cmdfail_and_exit_test.py
```

- 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 points here

- Use gback_glob_test.py as a starter.

Additional Tests (Part 2)

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



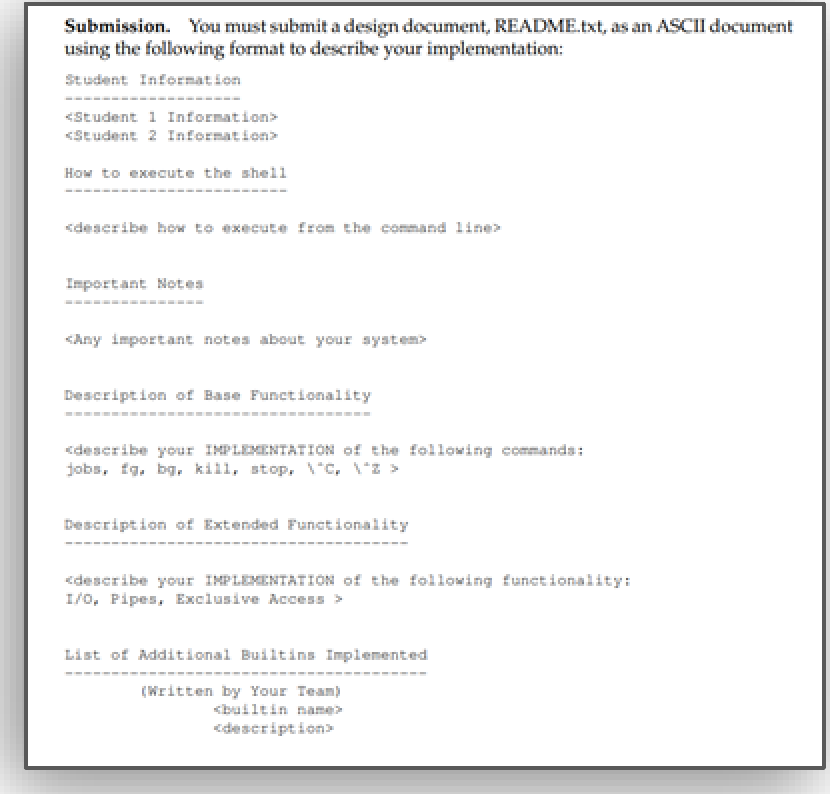
```
$ file custom.tst
```

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

- If it includes Windows line terminators (CR, CRLF, etc) it will fail
- We want \n, not \r\n

Design Document

- When you submit you must include a README.txt describing your implementation of P1
- Explain the custom built-ins created and approach taken to develop them.
- TAs will assign credit only for the functionality for which test cases and documentation exist



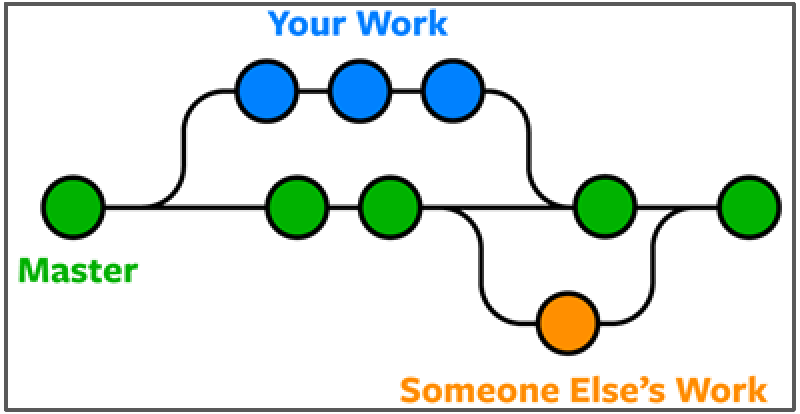
Submission. You must submit a design document, README.txt, as an ASCII document using the following format to describe your implementation:

```
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 and Gitlab 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
```

- Revert to the previous commit:

```
$ git reset [--hard]
```

Basic Git Commands

- Create a new branch from the current branch:

```
$ git checkout –b <new_branch_name>
```

- Switch to another branch:

```
$ git checkout <branch_name>
```

- Merge a branch into the current branch

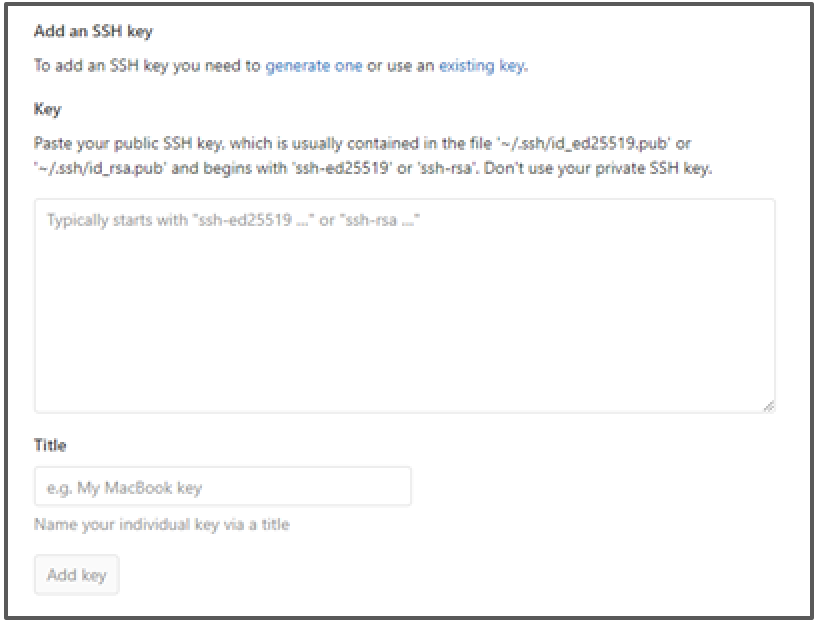
```
$ git merge <branch_name>
```

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

e.g. My MacBook key

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.
```

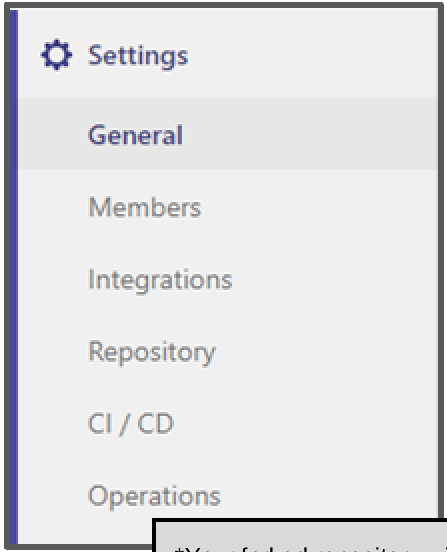
- You can get in-depth explanations here:
  - Generate a key
  - Use an existing key

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



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

IMPORTANT: Set forked repository to **private**

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

The GNU Project Debugger (GDB)

Starting GDB

- 1. Invoke GDB with a program:

```
$ gdb program
```

- 2. Run with command

```
(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>
```

Light reading: https://visualgdb.com/gdbreference/commands/set_follow-fork-mode

Valgrind

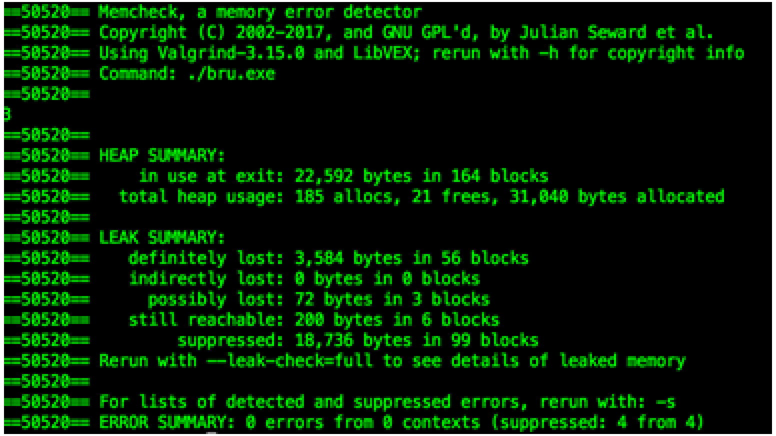
1. Start valgrind

```
$valgrind --log-file="output.log" --leak-check=full ./cush
```

2. Run commands in your shell

```
cush@cedar in src> jobs
cush@cedar in src> ls
```

3. Check the log file



```
==50520== Memcheck, a memory error detector
==50520== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==50520== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==50520== Command: ./bru.exe
==50520==
3
==50520==
==50520== HEAP SUMMARY:
==50520==     in use at exit: 22,592 bytes in 164 blocks
==50520==   total heap usage: 185 allocs, 21 frees, 31,040 bytes allocated
==50520==
==50520== LEAK SUMMARY:
==50520==    definitely lost: 3,584 bytes in 56 blocks
==50520==    indirectly lost: 0 bytes in 0 blocks
==50520==      possibly lost: 72 bytes in 3 blocks
==50520==    still reachable: 200 bytes in 6 blocks
==50520==         suppressed: 18,736 bytes in 99 blocks
==50520== Rerun with --leak-check=full to see details of leaked memory
==50520==
==50520== For lists of detected and suppressed errors, rerun with: -s
==50520== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

Advice

Advice

- **START EARLY**
- READ! Then create a roadmap before coding
- 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**
  - TA’s all have unique ways of implementing/coding. You will hear different answers that both work.
  - 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
  - In-Person Meetings, Discord, Zoom, etc.

Sources

- Referred to previous help session slides created by previous UTA's Kent McDonough, Connor Shugg, Joe D'Anna, Chris Cerne, Justin Vita, Sam Lightfoot, and Alex Kyer, Timothy Wu, Tanvi Allada, Vineet Marri, and Zhuowei Wen since the Fall 2023 Semester
- Spencer Keefer created the revised slides

Thanks for attending!

Questions?