CSE 15L: Software Tools and Techniques Laboratory

Winter 2021 - http://ieng6.ucsd.edu/~cs15x

Instructors: Gary Gillespie Keith Muller

Class sessions will be recorded and made available to students asynchronously.

Final Exam Format

- Exam will be on Canvas (canvas.ucsd.edu)
- Available to start any time on either assigned day
  - Select any CSE 15L exam (Monday or Friday)
  - Start exam from 12:01 am to 11:59 pm (Pacific Time)
  - Once started, you have 90 min to complete that exam
  - Can't select another exam once you've started an exam
- Questions will be multiple-choice, fill-in-the-blank, matching, and short answer (including writing short functions in Bash)
- Open book, open note, open Internet but no discussing the exam with other students until all finals have ended
- You may use a Bash terminal while taking the exam, but it is not required for any question

How I would Study: Step 1
Create a List of Important Topics

- Copying files: local and remote
- File name expansion
- Working directory modification navigation: pwd; cd ..; cd dir; cd ~; cd ~user
- Know the commands: ls, touch, cat, echo, mkdir, mv, man, sort, uniq, grep, ps, test, expr,ssh, scp, sftp, …
- script parameters
- using pipes
- file redirection >,<,>>, noclobber differences
- File access rights and all forms of chmod
- bash job control:fg,bg, background jobs, cntrl-z, etc.
- shell args
- shell control flow (if, loops, case,variable compare (ints and strings),…)
- shell functions
- ant build.xml, Makefile
- git commands:
- java logging
- jdb
- virtualization (containers versus vm's)

How I would Study: Step 2
Play the Role of the Instructor

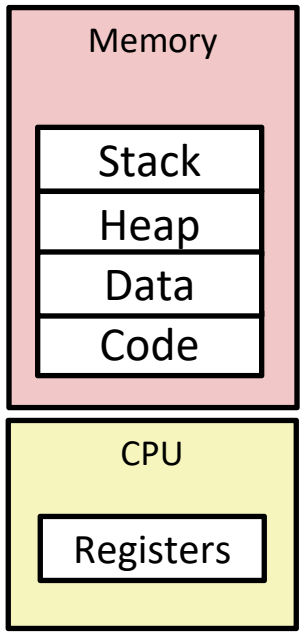
Write Your own Exam Questions and Answer Key for Each Topic

Processes

- Informal definition:

  A process is a program in execution.

- Process is not the same as a program.
  - Program is a passive entity stored in disk
  - Program (code) is just one part of the process.
- How to start a process?
  - Execute a utility, program, or script!

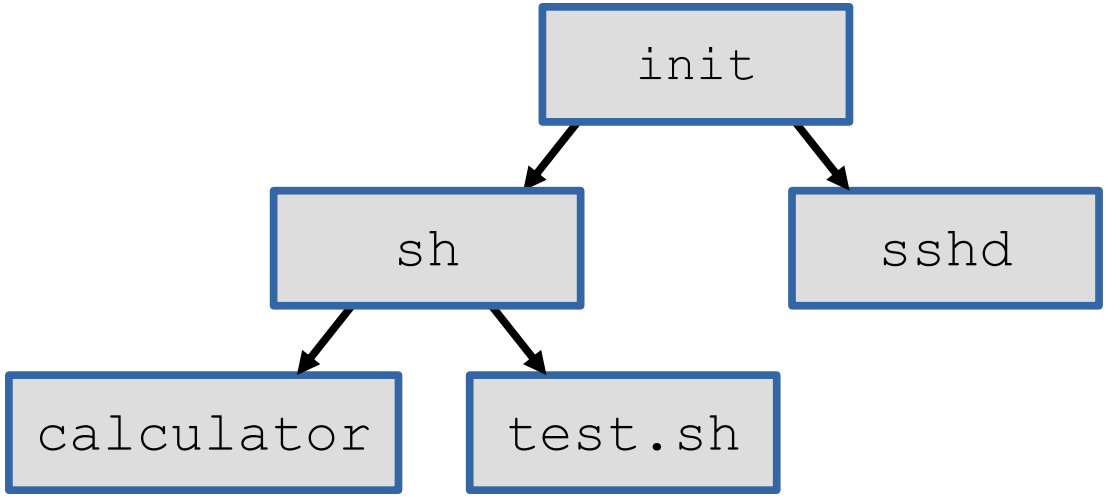


Unix Processes

- Every process has a parent, forming a hierarchy
  - PID: Process IDentifier
  - PPID: Parent Process Identifier
- Grandparent of all processes is init (PID 1)
- Orphaned processes are adopted by init
- On many linux systems a ppid of 0 is used as a placeholder indicating a process has no parent (init:Linux, launchd: MacOS)

Practice Problem 1

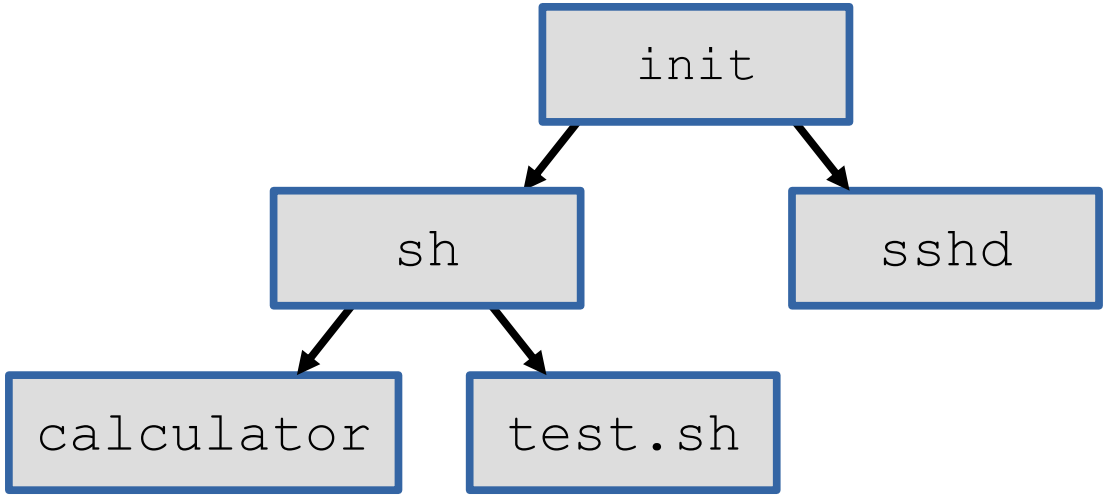
Using the following process hierarchy, fill in the output of **`ps -aef`**



UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	_A__	0	Dec22	?	00:00:01	_B_____
root	526	_C__	0	Dec22	?	00:00:00	sshd
cs15	_D__	_E__	0	Dec25	?	00:00:05	_F_____
cs15	1029	630	0	Dec25	?	00:00:00	_G_____
Cs15	1034	_H__	9	Dec25	?	00:00:15	test.sh

Practice Problem 1 - Solution

Using the following process hierarchy, fill in the output of **`ps -aef`**



UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	Dec22	?	00:00:01	init
root	526	1	0	Dec22	?	00:00:00	sshd
cs15	630	1	0	Dec25	?	00:00:05	sh
cs15	1029	630	0	Dec25	?	00:00:00	calculator
Cs15	1034	630	9	Dec25	?	00:00:15	test.sh

Foreground vs Background

- Foreground:
  - Processes executed on terminal run in foreground by default (Receive input from keyboard, Send output to screen)
  - Cannot run another command until previous command finishes
- Background:
  - For processes that don't require keyboard input, can send to background to run in parallel with **&**
  - example below runs two processes cocurrently

```
$ a.out & grep -r "pwd" . > pwd.txt &
[1] 18839
[2] 18876
...
[2]+ Done      grep -r "pwd" . > pwd.txt
```

Managing Foreground/Background

- **`jobs <-l>`**
  - Lists all background jobs in current shell, **+** indicates default for **`fg`**/**`bg`** command
- **`fg`** <#> or fg %<#>
  - Brings job number <#> to the foreground for keyboard input
- Ctrl-Z
  - Suspends current foreground job temporarily, placing it in a stopped condition
- **`bg`** <#> or bg %<#>
  - Begins running suspended job <#> in the background

Practice Problem 2

Using the jobs created by the following commands and acting fast in a single terminal window

what sequence of commands is required to get `autopilot.sh` running in the background and prevent "`Blue Wire`" from displaying on the terminal?

```
$ (sleep 10; echo "Red Wire") & (sleep 10; echo "Blue Wire") &
[1] 17510
[2] 17511
$ ./autopilot.sh
```

Practice Problem 2 - Solution

Using the jobs created by the following commands and acting fast in a single terminal window

what sequence of commands is required to get `autopilot.sh` running in the background and prevent displaying "`Blue Wire`" on the terminal?

```
$ (sleep 10; echo "Red Wire") & (sleep 10; echo "Blue Wire") &
[1] 17510
[2] 17511
$ ./autopilot.sh
```

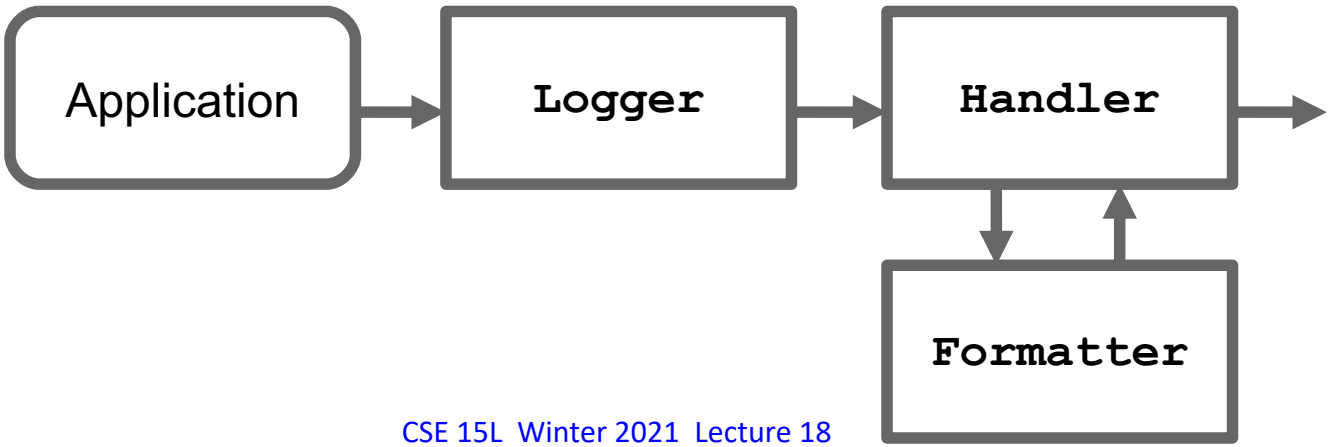
<Ctrl-Z> (Stops current foreground program ./autopilot.sh)

bg or **bg 3** (re-Start autopilot.sh in background, bg 3)

kill %2 or **kill 17511** (by default, `kill` will terminate a program. `kill %<n>` references by job. `kill <n>` by PID)

java.util.logging Structure

- Important classes in the java.util.logging package:
  - `Logger`
  - `Handler`
    - `ConsoleHandler, FileHandler, SocketHandler`
  - `Formatter`
    - `SimpleFormatter, XMLFormatter`
  - `Level`



Logging Levels

- The **Level** class
  - contains constants to specify the importance level of log messages and to control which log records are logged
- From highest importance to lowest:

```
Level.SEVERE
Level.WARNING
Level.INFO
Level.CONFIG
Level.FINE
Level.FINER
Level.FINEST
```

- Can be set in `properties.config` file to change logging functionality without recompiling code

Practice Problem 3

```
   ...
 3 public class LostInSpace {
 4   protected static Logger logger = Logger.getLogger("LostInSpace");
 5
 6   public static void main(String argv[])  {
 7     int[] countdown = {3, 2, 1};
 8
 9     logger.log(Level.WARNING, "Danger, Will Robinson!");
10     try {
12       int index = Character.getNumericValue(System.in.read());
13       System.out.println(countdown[index]);
14     }
15     catch (Exception ex) {
16       logger.severe("Problem!!");
17       return;
18     }
19     logger.config("You're safe. Goodbye.");
20   }
21 }
```

a.) List at least one way that an exception can be thrown from lines 12-13.
Note: Line 12 takes single character input from user and converts it to an integer.

b.) Which logging messages will be printed for a logger and handler level of Level.INFO and user input of 0?

c.) Which logging messages will be printed for a logger and handler level of Level.CONFIG and a user input of 4?

Practice Problem 3 - Solution

```
...
 3 public class LostInSpace {
 4   protected static Logger logger = Logger.getLogger("LostInSpace");
 5
 6   public static void main(String argv[])  {
 7     int[] countdown = {3, 2, 1};
 8
 9     logger.log(Level.WARNING, "Danger, Will Robinson!");
10     try {
12       int index = Character.getNumericValue(System.in.read());
13       System.out.println(countdown[index]);
14     }
15     catch (Exception ex) {
16       logger.severe("Problem!!");
17       return;
18     }
19     logger.config("You're safe. Goodbye.");
20   }
21 }
```

a.) List at least one way that an exception can be thrown from lines 12-13.
Note: Line 12 takes single character input from user and converts it to an integer.
User can input a non-numeric character -or- user can input a value greater than 2.

b.) Which logging messages will be printed for a logger and handler level of Level.INFO and user input of 0?
"Danger, Will Robinson!"

c.) Which logging messages will be printed for a logger and handler level of Level.CONFIG and a user input of 4?
"Danger, Will Robinson!" and "Problem!!"

Practice Problem 4

```
1  #!/bin/bash
2  binofnumbers=("1" "6" "3" "5" "9" "4")
3  output="pi"
4  echo "output="
5  output=".${binofnumbers[0]}"
6  echo $output
7  output="${binofnumbers[2]}${output}"
8  output="${output}${binofnumbers[5]}"
9  echo $output
```

What is printed to the terminal on line 4?

What is printed to the terminal on line 6?

What is printed to the terminal on line 9?

Practice Problem 4 - Solution

```
1  #!/bin/bash
2  binofnumbers=("1" "6" "3" "5" "9" "4")
3  output="pi"
4  echo "output="
5  output=".${binofnumbers[0]}"
6  echo $output
7  output="${binofnumbers[2]}${output}"
8  output="${output}${binofnumbers[5]}"
9  echo $output
```

What is printed to the terminal on line 4?

output=

What is printed to the terminal on line 6?

.1

What is printed to the terminal on line 9?

3.14

Practice Problem 5

Complete this code using a bash **`case`** statement.

```
# If the user entered "y" or "Y", then go to the next
# iteration of the loop.
# For any other input, exit from the loop (but not the program).
while (( 0 != 1 )); do
    echo "Run again?"
    read input

    # TODO: Enter your complete case statement below.
    # Use as many lines as you need to input the correct
    # case statement.
    --- enter ---
    --- your  ---
    --- code  ---
    --- here  ---

done
```

Practice Problem 5 - Solution

Complete this code using a bash **`case`** statement.

```
# If the user entered "y" or "Y", then go to the next
# iteration of the loop.
# For any other input, exit from the loop (but not the program).
while (( 0 != 1 )); do
    echo "Run again?"
    read input

    # TODO: Enter your complete case statement below.
    # Use as many lines as you need to input the correct
    # case statement.
    case "$input" in
        y)
            continue;;
        Y)
            continue;;
        *)
            break;;
    esac
done
```

Makefile timestamps and dependencies

- The dependencies in a rule can be the names of other targets in the Makefile
- In that case, a rule like

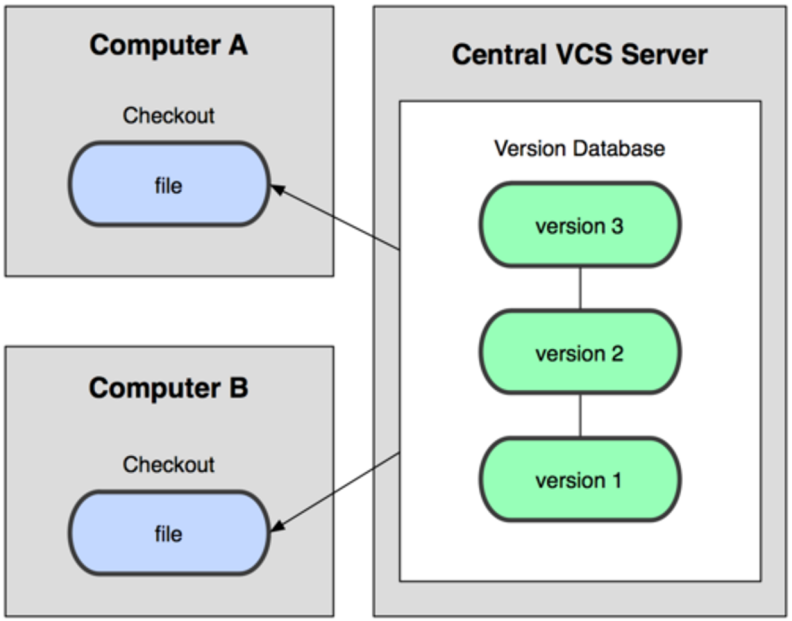
```
target0 : target1 … targetM
        action1
        …
        actionN
```

will execute the actions for **target$_1$** through **target$_M$**, and then the actions for **target$_0$**

- The actions will be performed if any of the dependencies are newer than the target.

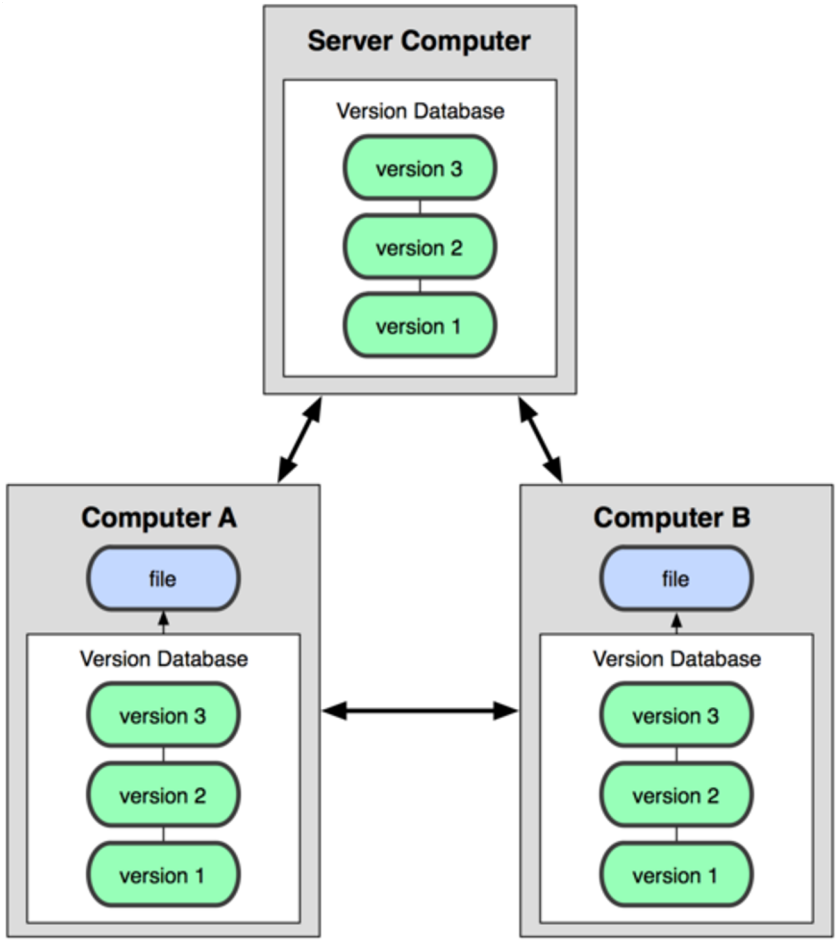
Git uses a distributed model

Centralized Model



(CVS, Subversion, Perforce)

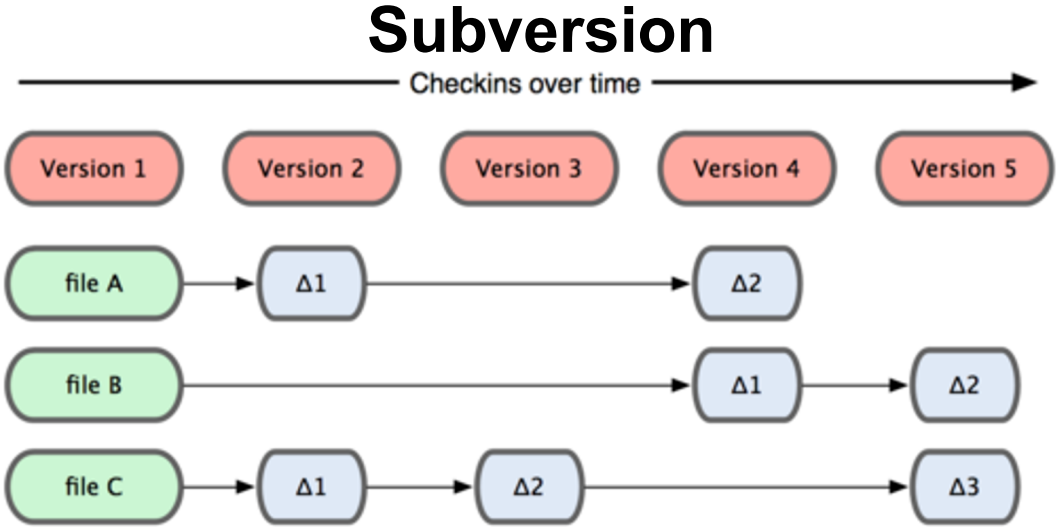
Distributed Model

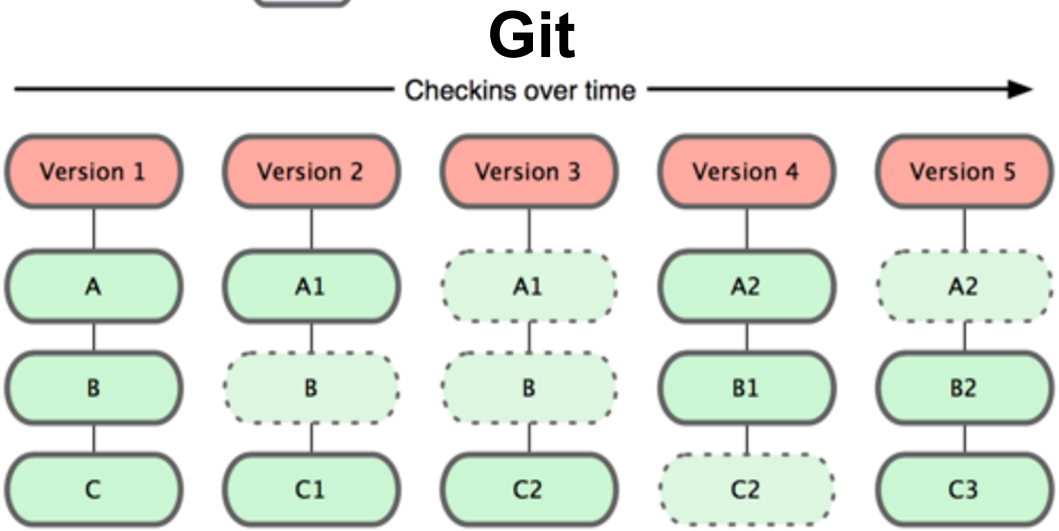


(Git, Mercurial)

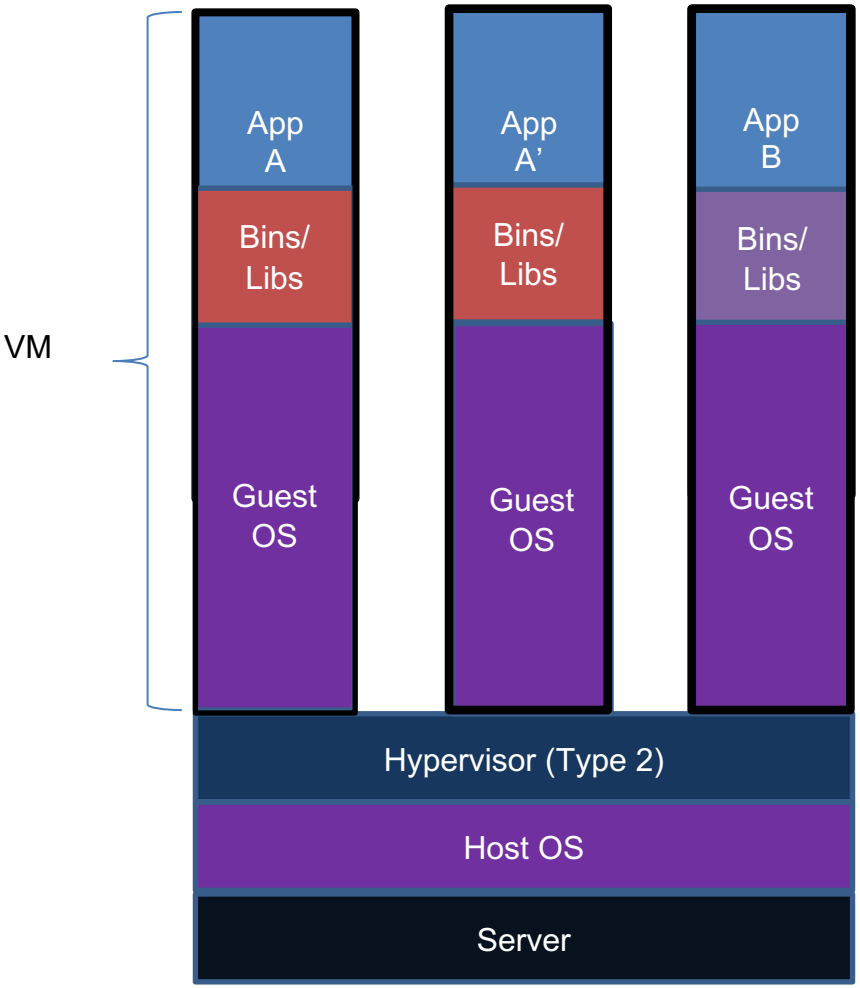
Result: Many operations are local

Git takes snapshots

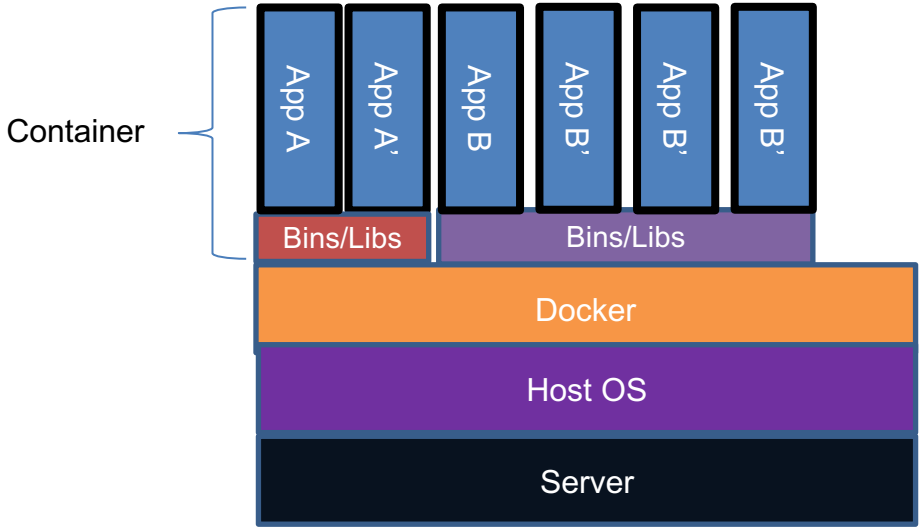




Containers vs VMs



Containers are isolated, but share OS and, were appropriate, bins/libraries



Expansion and Wildcards

- Each time we type a command and press the enter key, bash performs several substitutions upon the text before it carries out our command

```
$ echo this is a test
this is a test
```

- pathname expansion

```
$ echo *
Desktop fooDocuments output.txt fooMusic Pictures
Public Templates Videos
$ echo foo*
DfooDocuments fooMusic
```

Positional Parameters

- When a shell script is invoked with a set of command line parameters/arguments each of these parameters are copied into special variables that can be accessed

  - **`$0`** This variable that contains the name of the script
  - **`$1`**, **`$2`**, … **`$9`** 1st, 2nd, and 9th command line parameter
  - **`${1}`**, **`${2}`**, …, **`${10}`** 1st, 2nd, and 10th argument parameter
  - **`$#`** Number of command line parameters
  - **`$$`** process ID of the shell
  - **`$@`** same as **`$*`** but as a list one at a time
  - **`$?`** Return code 'exit code' of the last command
  - **`shift`** command: This shell command shifts the positional parameters by one towards the beginning and drops **`$1`** from the list.
    - After a **`shift`**, **`$2`** becomes **`$1`** and so on.
    - It is a useful command for processing the input parameters one at a time.

Review: Redirecting and Piping

- Redirecting via angle brackets **<**, **>**, **<<**, **>>**
  - Redirecting standard input and output follows a similar principle to that of piping except that redirects with files, not commands.
- Piping **|**
  - An important early development in Unix , a way to pass the output of one tool to the input of another.

EXAMPLE

```
$ touch file1                  # ensure file1 is present
$ echo Hello > file1           # file1 got clobbered!

$ set –o noclobber             # prevent clobbering!
$ echo Hello > file1
-bash: file1: cannot overwrite existing file
$ echo Hello >> file1          # noclobber allows append!

$ set +o noclobber             # allow clobbering!
$ echo Hello > file1           # file1 clobbered again!
```

`test` Options for File Inquiry

`[ -d filename ]`	Test if filename is a directory
`[ -f filename ]`	Test if filename is not a directory
`[ -s filename ]`	Test if filename has nonzero length
`[ -r filename ]`	Test if filename is readable
`[ -w filename ]`	Test if filename is writable
`[ -x filename ]`	Test if filename is executable
`[ -o filename ]`	Test if filename is owned by the user
`[ -e filename ]`	Test if filename exists
`[ -z filename ]`	Test if filename has zero length

All these conditions return true (0) if satisfied and false (1) otherwise.

All the file test conditions include an implicit test for existence

example: you do not need to test if a file exists and is readable, It won't be readable if it doesn't exist.

Combining Tests

&& represents AND
|| represents OR

Syntax:

```
if cond1 && cond2 || cond3 …
```

An alternative form is to use a compound statement in **test** using **–a** and **–o** keywords. For example:

```
if [ cond1 –a cond2 –o cond3 … ]
```

where **cond1**, **cond2**, **cond3** are either commands returning a value or test conditions of the form:

```
test arg1 op arg2
```
or
```
[ arg1 op arg2 ]
```

Arithmetic Operators

- If variables are strings, how do we do math?
- Several methods in bash!

  For the variable **`xyz=2020`**

  - **`expr`** command `$ xyz=`expr $xyz + 1``
  - **`let`** command `$ let "xyz=xyz + 1"`
  - **`(( … ))`** syntax `$ xyz=$((xyz + 1))`

- Note: All methods are for integer math only! If you need more precision, look up **`bc`**

Using `expr`

- **`expr`** command supports the following operators:
  - see %man expr
  - arithmetic operators: +, -, *, /, %
  - comparison operators: <, <=, ==, !=, >=, >
  - Boolean/logical operators: &&, ||
  - parentheses: (…)
  - precedence is the same as C, Java

- **`expr`** takes values (numerical or evaluated variable **`$myvar`**) and operators as arguments

- **Caution!**
  - All arguments must be separated by spaces!!
  - To multiply, you must use * and not *

The Last Step….

- Exam will be on Canvas (canvas.ucsd.edu)
- Available to start any time on either assigned day
  - Select any CSE 15L exam (Monday or Friday)
  - Start exam from 12:01 am to 11:59 pm (Pacific Time)
  - Once started, you have 90 min to complete that exam
  - Can’t select another exam once you’ve started an exam
- Questions will be multiple-choice, fill-in-the-blank, matching, and short answer (including writing short functions in Bash)
- Open book, open note, open Internet but no discussing the exam with other students until all finals have ended
- You may use a Bash terminal while taking the exam, but it is not required for any question