

# CS 35L

LAB 8

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

- Course Info
- OS
- File system
- Basic commands
- Lab Assignment



# Course Info

# What is this course about?

“Fundamentals of commonly used software tools and environments, particularly open-source tools to be used in upper division computer science courses.”

NO TEXTBOOK :)



# Course Organization

- Lab assignments and homeworks
- Monday session - Lab assignment
- Wednesday session - Homework
- 10 lab assignments and 10 homeworks



# Homeworks

- Submit on CCLE
- Assignments are due by 23:55 on the specified date (i.e., five minutes before midnight at the end of the day).
- Assignments 2–9 and their schedule are tentative.
- Assignments are subject to change. Any changes will be informed to you



# Assignment 10

- Read one of the stories referenced in recent issues of ACM TechNews
- Write a brief review of the story and submit on CCLE
- A suggested length is 400 to 1200 words per topic. Include tables, graphs, and images as appropriate.
- Mail me your topic and a tentative presentation date.
- Sign up will start in week 2
- 3 - 4 presentations in every lab session



# Policy

- You are expected to do your homeworks by yourself.
- You can share ideas and discuss general principles with others in the class, but all the code and writings that you submit must be your own work; do not share them with others.
- Some assignments will be done on SEASnet GNU/Linux servers. In these cases, take care to not run commands like `su` and `sudo` that would make it appear to the system administrators that you might be trying to break into the system.
- Avoid arbitrary limits on the length or number of any data structure, including symbols, strings, and line length. It is OK to impose a non-arbitrary limit, e.g., because your computer runs out of memory or because of the limited range of the C `ptrdiff_t` type, but it is not OK to impose an arbitrary limit, e.g., a limit of at most 255 characters in a symbol.
- Please stick to coding styles used in the course material rather than inventing your own style.  
(Coding styles will be emailed)

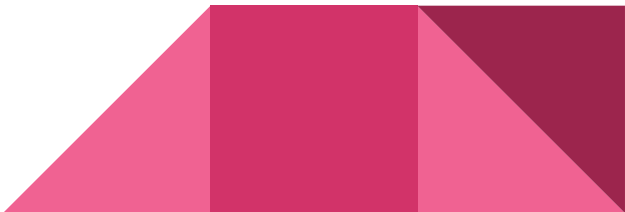


# Grading

Grades are weighted as follows:

- 50% homeworks (equally weighted)
- 50% final exam

PENALTY: “The lateness penalty for an assignment that is submitted between  $N$  and  $N+1$  full days late (where  $N$  is nonnegative) is  $2^N\%$  of the assignment's value. For example, if an assignment is worth 100 points, the penalty is 1 point for being up to 1 day late, 2 points for being from 1 to 2 days late, 4 points for being from 2 to 3 days late, and so forth. Assignments are not accepted after the last day of instruction (this is typically the Friday before final exams), and are not accepted after the lateness penalty renders them irrelevant to the final grade.”

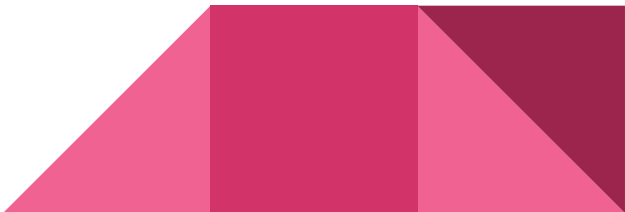


# Finals

Finals: **Monday, June 12, 2017, 3:00 PM - 6:00 PM**

## NO MAKEUP EXAMS

Students must follow the [UCLA Student Conduct Code](#), which prohibits cheating, fabrication, multiple submissions, and facilitating academic dishonesty. A summary of the academic integrity material of the Student Conduct Code can be found in the [Student Guide to Academic Integrity](#), and the [Office of the Dean of Students](#) has a [workshop on academic integrity](#).





# Theory

# OS History

- Till 1960's no operating system(OS)
- Every program needed
  - the full hardware specification to run correctly and perform standard tasks,
  - own drivers for peripheral devices
- Punched cards with programs and data. Later libraries with support code on punched cards
- Monitor programs or Run time libraries - started before the job, read in the customer job, control its execution, record its usage, reassign hardware resources after the job ended, and immediately go on to process the next job.
- OS - heavily dependent on machine hardware until System/360 which had same instruction and I/O architecture

# Multiuser and MultiProcess OS

Multiple users are allowed access to the resources

Batch processing vs Time sharing system



# CLI vs GUI

## CLI

Steep learning curve

Pure control (e.g., scripting)

Cumbersome multitasking

Speed: Hack away at keys

Convenient remote access

## GUI

Intuitive

Limited Control

Easy multitasking

Limited by pointing

Bulky remote access

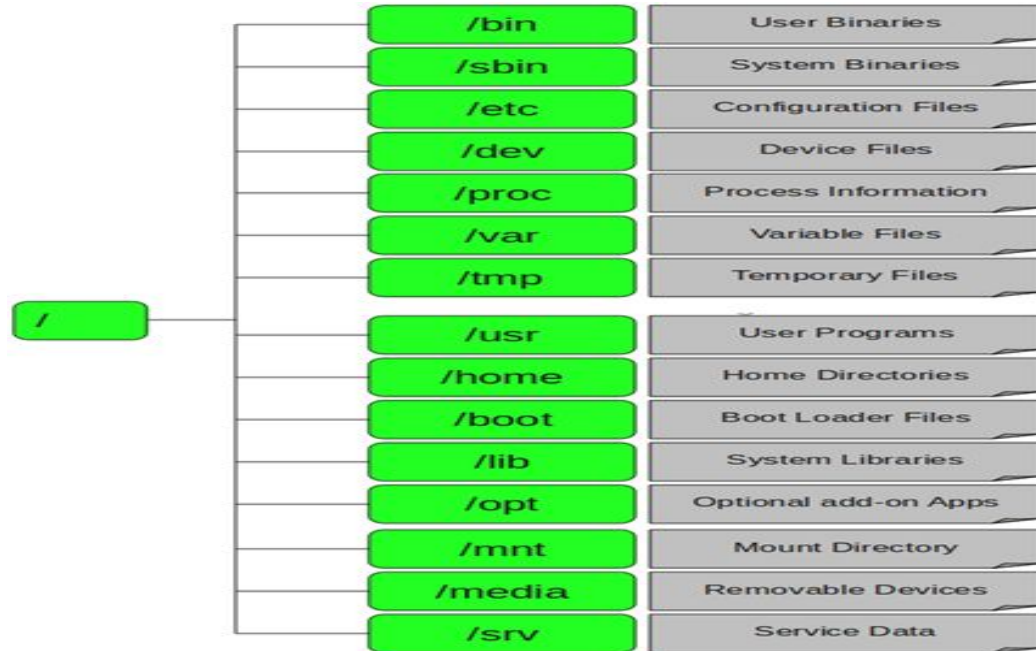


# Files

- In Unix, everything is either a File or a Process
- Process - program that is being executed
- File - Collection of data
  - *Directories*: files that are lists of other files.
  - *Special files*: the mechanism used for input and output. Most special files are in /dev, we will discuss them later.
  - *Links*: a system to make a file or directory visible in multiple parts of the system's file tree.
  - *Sockets*: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.
  - *Named pipes*: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.



# Linux File System





# Unix File Permissions

```
shum@sol:~$ ls -l
total 20
drwx----- 2 shum      staff    4096 Jan 16 22:04 Mail
drwx----- 3 shum      staff    4096 Jan 16 14:15 csc128
drwxr-xr-x  2 shum      staff    4096 Jan 13 16:42 public
drwxr-xr-x  2 shum      staff    4096 Jan 16 14:07 public_html
-rw-r--r--  1 shum      staff    628 Jan 15 20:04 verse
```

Diagram illustrating the components of the Unix file permissions output:

- file type**: Indicated by the first character (d for directory, - for regular file).
- number of hard links**: Indicated by the next two characters (2, 3, 2, 2, 1).
- user (owner) name**: Indicated by the next three characters (shum, shum, shum, shum, shum).
- group name**: Indicated by the next three characters (staff, staff, staff, staff, staff).
- size**: Indicated by the next four characters (4096, 4096, 4096, 4096, 628).
- date/time last modified**: Indicated by the next eight characters (Jan 16 22:04, Jan 16 14:15, Jan 13 16:42, Jan 16 14:07, Jan 15 20:04).
- filename**: Indicated by the last six characters (Mail, csc128, public, public\_html, verse).
- permissions**: Indicated by the last nine characters (drwx-----, drwx-----, drwxr-xr-x, drwxr-xr-x, -rw-r--r--).
- permissions breakdown**:
  - rwx**: Readable (r), Writeable (w), Executable (x).
  - other (everyone) permissions**: Indicated by the last three characters (---, ---, -r-x).
  - group permissions**: Indicated by the next three characters (---, ---, r-x).
  - user permissions**: Indicated by the first three characters (drw, drw, drwx, drwx, -rw).

# Basics

# Getting started

## 1) SEAS Server – best option, highly recommend

- Lnxsrv.seas.ucla.edu (use Inxsrv06, Inxsrv07 or Inxsrv09)
- Make sure your PATH environment variable has /usr/local/cs/bin
- export PATH=\$PATH:/usr/local/cs/bin

## 2) On your computer

- Install or try Ubuntu
- Run with Windows (<https://wiki.ubuntu.com/WubiGuide>)
- Easy to remove Ubuntu from Windows via Control Panel, if you don't need Ubuntu.

# Getting started

## 3) Virtual Machine

- VMWare
- Virtual Box

## 4) Live CDs on BH3760 Computers

- Don't install Ubuntu, try Ubuntu

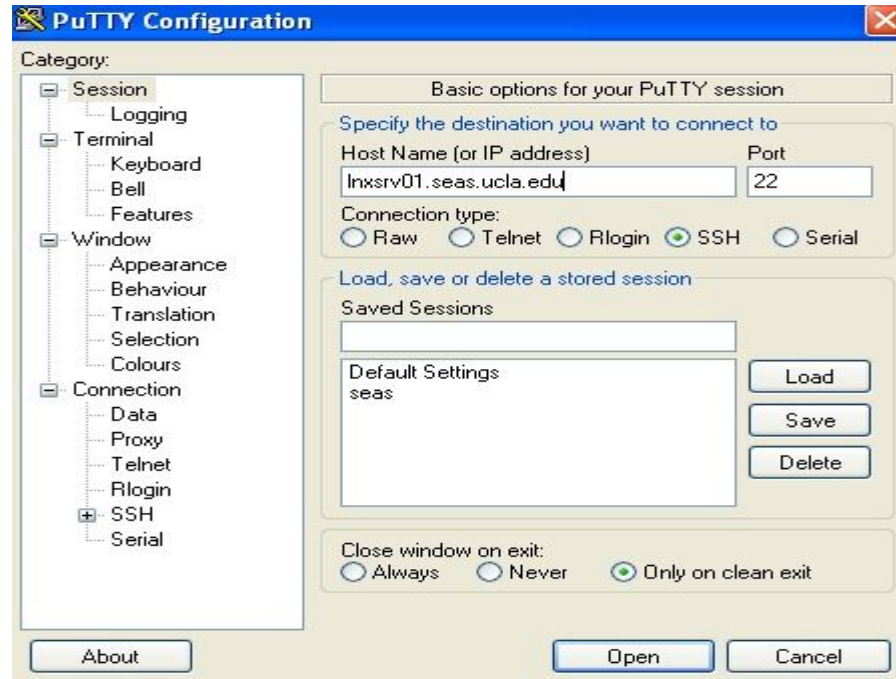


# Connecting to SEAS from Windows

- Use Putty
- Host name: `lnxsrv.seas.ucla.edu`
- User name and Password : your SEAS username and password



# Connecting to SEAS from Windows



# Connecting to SEAS from OS X or Linux

```
ssh username@lnxsrv.seas.ucla.edu
```

Username = your SEAS username



# Moving Around

pwd - print working directory

cd - change working directory

- ~ - home directory
- . - current directory
- / - root directory, or directory separator
- .. - parent directory





# Commands

- mv: move a file (no undos!)
- cp: copy a file
- rm: remove a file
- mkdir: make a directory
- rmdir: remove a directory
- ls: list contents of a directory
  - -d: list only directories
  - -a: list all files including hidden ones
  - -l: show long listing including permission info
  - -s: show size of each file, in blocks



# Changing file attributes

Chmod - change access permissions

- read (r), write (w), executable (x), special execute (X), setuid/gid (s), sticky (t)
- User, group, others



# Changing file attributes

Reference	Class	Description
u	user	the owner of the file
g	group	users who are members of the file's group
o	others	users who are not the owner of the file or members of the group
a	all	all three of the above, is the same as <i>ugo</i>

# Changing file attributes

Operator	Description
+	adds the specified modes to the specified classes
-	removes the specified modes from the specified classes
=	the modes specified are to be made the exact modes for the specified classes

Mode	Name	Description
r	read	read a file or list a directory's contents
w	write	<b>w</b> rite to a file or directory
x	execute	<b>e</b> xecute a file or recurse a directory tree

# Changing file attributes

#	Permission
7	full
6	read and write
5	read and execute
4	read only
3	write and execute
2	write only
1	execute only
0	none

# Changing file attributes

## Usage

– `chmod ["references"]["operator"]["modes"] "file1" ...`

## Example:

- `chmod ug+rw mydir`
- `chmod a-w myfile,`
- `chmod ug=rx mydir`
- `chmod 664 myfile`



# man

Man pages - user manual

Man command - Format and display man pages

For info about man: man man



# Look these up

- cat
- Head, tail
- ls
- du
- ps
- kill
- diff
- cmp
- wc
- sort
- which





# Lab Assignment

<http://web.cs.ucla.edu/classes/spring17/cs35L/assign/assign1.html>

# Next class

- Links
- Emacs
- Homeworks



# CS 35L

LAB 8, Session 2

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

- Symbolic Links and Hard links
- Emacs
- Lab Assignments and Homework





# Links

# Symbolic Links (soft links)

**symbolic link** - the name of a file that contains a reference to another file or directory, either in the form of an absolute path or relative path

A text string that is interpreted as a path to another file or directory ( called the target )

If target is nonexistent then link is broken, orphaned or dead.



# Hard link

Directory entry associating a name with a file

Equivalent to giving one file, multiple names

Creates an alias effect

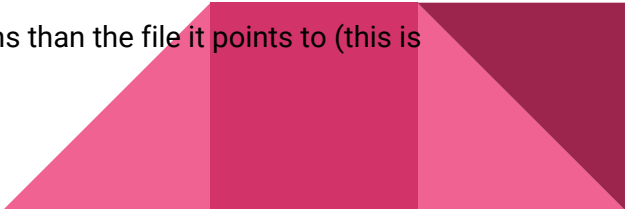


# Hard links vs soft links

## Hard links:

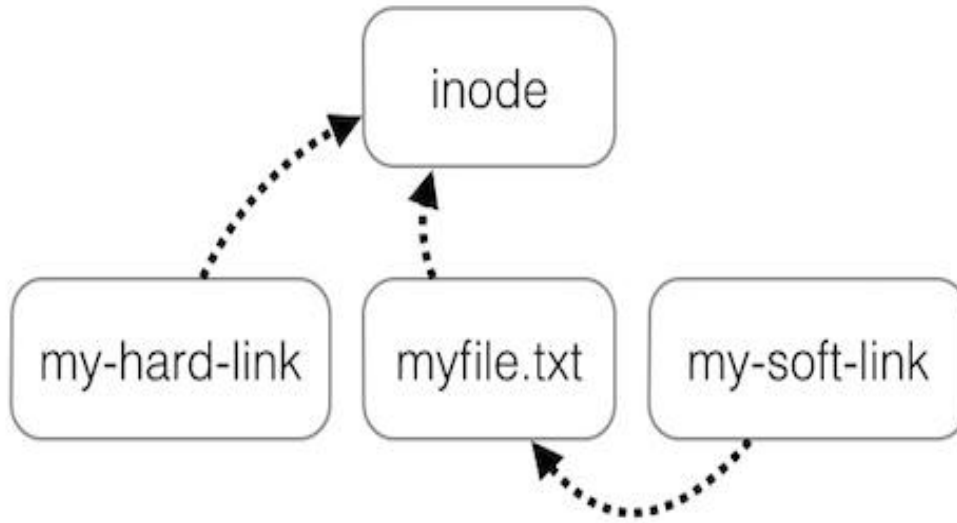
- indistinguishable from other directory entries, because every directory entry is hard link
- "original" can be moved or deleted without breaking other hard links to the same inode
- only possible within the same filesystem
- permissions must be the same as those on the "original" (permissions are stored in the inode, not the directory entry)
- can only be made to file, not directories

## Symbolic links (soft links):

- simply records that point to another file path. (ls -l will show what path a symlink points to)
  - will break if original is moved or deleted. (In some cases it is actually desirable for a link to point to whatever file currently occupies a particular location)
  - can point to a file in a different filesystem
  - can point to a directory
  - on some file system formats, it is possible for the symlink to have different permissions than the file it points to (this is uncommon)
- 



# Hard links vs soft links



# Link Creation

```
cd ~/Documents/
```

```
touch file1.txt
```

```
touch file2.txt
```

```
echo "file1" > file1.txt
```

```
cat file1.txt
```

```
echo "file2" > file2.txt
```

```
cat file2.txt
```



# Link Creation

- **ln file1.txt hardlink**
- **cat hardlink**
- **ln -s file2.txt softlink**
- **cat softlink**
- **ls -ali**
- **rm file2.txt**
- **ls -ali**
- **cat softlink**
- **rm file1.txt**
- **ls -ali**
- **cat hardlink**



# Basic operations

# Creating, copying, moving and deleting a file

```
touch myfile1.txt
```

```
cp myfile1.txt myfile2.txt
```

```
ls -l
```

```
mv myfile1.txt myfile3.txt
```

```
ls -l
```

```
rm myfile3.txt
```

```
ls -l
```



# Creating and Deleting directory

**mkdir mydir**

**rmdir mydir**





# Emacs

# Creating a file and adding content

**Emacs myfile.txt**

**Save file: C-x C-s**

**Exit Emacs: C-x C-c**

**Quit (i.e. interrupt) command: C-g**





# Copy and pasting content in a file

**Set a mark (select a region of text you want to copy/cut) : C-space**

**Copy : M-w**

**Cut : C-w**

**Paste : C-y**

**Delete line: C-k (puts it into clipboard)**

**Read only buffer? Clear by C-x C-q**



# Directory editor

**Search for a word: C-s (forward), C-r (to reverse)**

**Enter mode by: C-x d**

**Allows you to operate on files: remove, rename, encrypt, decrypt, edit**



# Running shell commands

**M-x shell (interactive shell)**



# Building programs

**Compile programs: M-x compile**

**Then, specify command to compile**

**Tip for homework: `gcc hello.c -o hello`**

**Run the executable by running the shell command (M-x shell)**

**`./hello`**



# Homework



# CS 35L

LAB 8, Session 2

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

- Pipelines and redirection
- Shell script
- Interpreted languages



# Details

`mkdir lab2_1` (Make a directory for each lab session)

`cd lab2_1`

`touch lab.log` (optional)

`touch lab.txt`

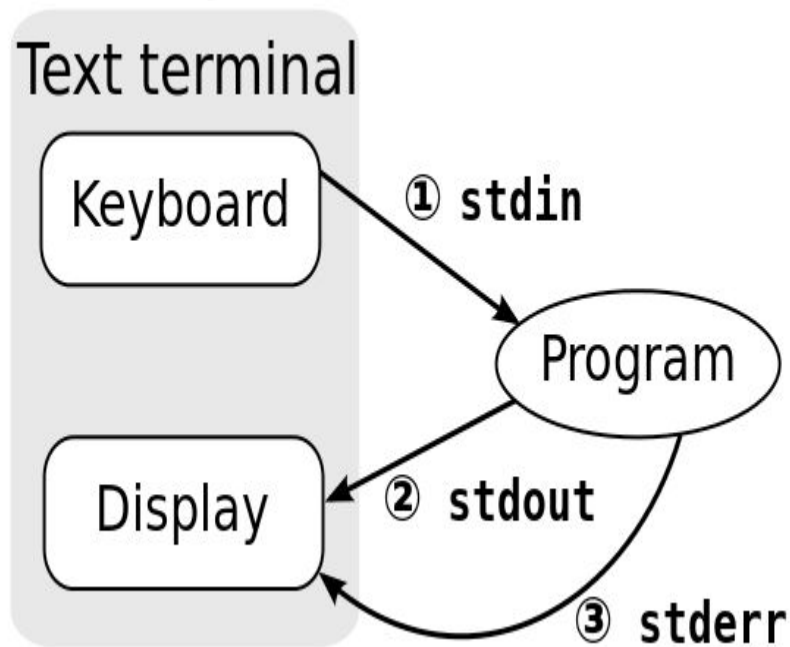
`touch hw.txt`





# Redirection and Pipeline

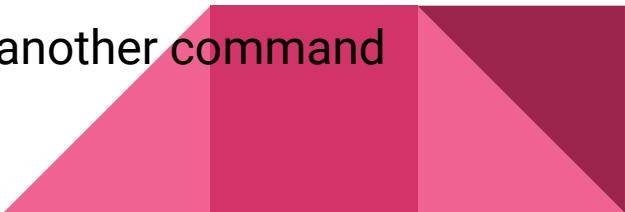
# Standard Streams



Every program has these 3 streams to interact with the world

- **stdin (0):** contains data going into a program
- **stdout (1):** where a program writes its output data
- **stderr (2):** where a program writes its error msgs

# Redirection - Bourne shell

- `>` Redirect standard output
  - `2>` Redirect standard error
  - `2>&1` Redirect standard error to standard output
  - `<` Redirect standard input
  - `|` Pipe standard output to another command
  - `>>` Append to standard output
  - `2>&1|` Pipe standard output and standard error to another command
- 

# Redirect standard output

```
ls -l
```

```
ls -l > file1.txt
```

```
cat file1.txt
```

```
rm file1.txt
```



# Redirect standard input

```
echo "Hello"
```

```
echo "Hello" > myfile.txt ( output redirection)
```

```
cat < myfile.txt (input redirection)
```

```
echo "Appending another Hello" >> myfile.txt
```

```
cat < myfile.txt
```



# Pipeline

`find . -type f` (Find all files in current directory and subdirectories)

`wc -l` (prints new line counts)

`find . -type f | wc -l` (show total number of files in current directory and subdirectories)



# Shell Script

# Shell?

The shell is a user interface to access OS's services like file management, process management etc

Accepts commands as text, interprets them, uses kernel API to carry out what the user wants – open files, start programs etc and displays output





# Types of UNIX shells

Two major types of shells:

1. The Bourne shell - the default prompt is the \$ character. Subcategories - Bourne shell ( sh), Korn shell ( ksh), Bourne Again shell ( bash), POSIX shell ( sh)
2. The C shell - the default prompt is the % character. Subcategories - C shell ( csh), TENEX/TOPS C shell ( tcsh)



# Compiled vs. Interpreted

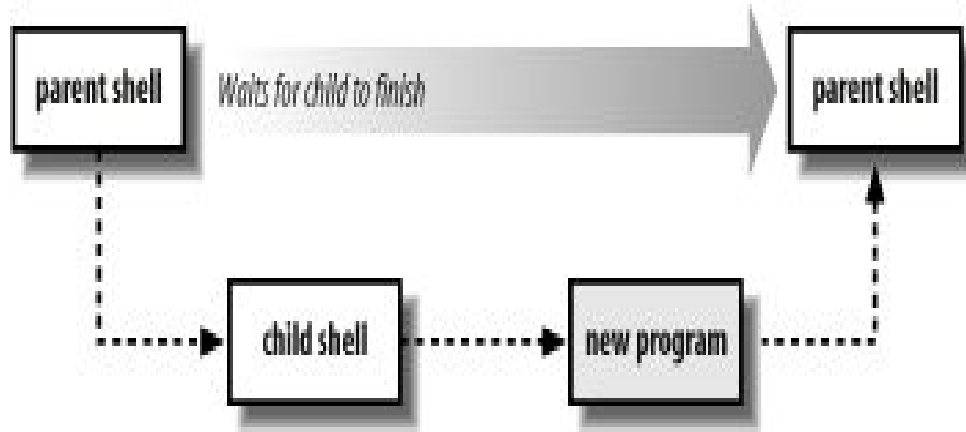
## Compiled

- Programs are translated from their original source code into machine code that is executed by hardware
- Efficient and fast
- Require recompiling
- Work at low level, dealing with bytes, integers, floating points, etc.
- Ex: C/C++

## Interpreted

- Interpreter program (the shell) reads commands, carries out actions commanded as it goes
- Much slower execution
- Portable
- High-level, easier to learn
- Ex: PHP, Ruby, bash

# Shell script executions



A shell script file is just a file with shell commands

When shell script is executed a new child “shell” process is spawned to run it

The first line is used to state which child “shell” to use:

```
#!/bin/sh
```

```
#!/bin/bash
```

# Simple shell script

1. `emacs hello.sh`

In the file,

2. `#!/bin/bash`

`echo "Hello"`

3. Save and exit (`C-x C-s`, `C-x C-c`)



# Executing a shell script

```
bash ./hello.sh
```

```
ls -l hello.sh
```

```
chmod +x hello.sh
```

```
ls -l hello.sh
```

```
bash ./hello.sh
```



# Output using echo and printf

echo writes arguments to stdout, can't output escape characters (without -e)

Example: `echo "Hello\nworld"` vs `echo -e "Hello\nworld"`

printf can output data with complex formatting, just like C printf()

Example: `printf "%.3e\n" 46553132.14562253`



# Quotes

## Three kinds of quotes

- Single quotes `' '`
  - Do not expand at all, literal meaning
- Double quotes `" "`
  - Almost like single quotes but expand `$`, `' '` and `\`
- Backticks `` ``
  - Expand as shell commands

Refer example program



# Declaring variables

Declared using =

```
var="hello"  #NO SPACES!!!
```

Referenced using \$

```
echo $var
```





# Decision statements

If statements

If..else statements

case..esac statements

Refer sample programs



# If statement

if [ expression ]

then

Statement(s) to be executed if expression is true

fi

if [ expression ]

then

Statement(s) to be executed if expression is true

else

Statement(s) to be executed if expression is not true

fi



# Case statement

case word in

pattern1)

Statement(s) to be executed if pattern1 matches

;;

pattern2)

Statement(s) to be executed if pattern2 matches

;;

pattern3)

Statement(s) to be executed if pattern3 matches

;;

esac



# For loop

```
for var in word1 word2 ... wordN  
do  
    Statement(s) to be executed for every word.  
done
```



# While loop

while command

do

Statement(s) to be executed if command is true

done



# Select loop

```
select var in word1 word2 ... wordN
```

```
do
```

```
    Statement(s) to be executed for every word.
```

```
done
```



# Special Variables

**\$0** - The filename of the current script.

**\$n** - These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is \$1, the second argument is \$2, and so on).

**\$#** - The number of arguments supplied to a script.

**\$\*** - If a script receives two arguments, \$\* is equivalent to \$1 \$2.

**\$?** - The exit status of the last command executed.

**\$\$** - The process number of the current shell. For shell scripts, this is the process ID under which they are executing.



# Exit values

Value	Typical/Conventional Meaning
-------	------------------------------

0	Command exited successfully.
---	------------------------------

> 0	Failure to execute command.
-----	-----------------------------

1-125	Command exited unsuccessfully. The meanings of particular exit values are defined by each individual command.
-------	---

126	Command found, but file was not executable.
-----	---

127	Command not found.
-----	--------------------

> 128	Command died due to receiving a signal
-------	--





# CS 35L

LAB 8, Session 2

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

- Unix wildcards, basic regular expressions
- Commands - Grep, sed
- Lab assignment



# Details

`mkdir lab2_2` (Make a directory for each lab session)

`cd lab2_2`

`touch lab.log` (optional)

`touch lab.txt`

`touch hw.txt`



# UNIX WILDCARDS

# Wildcard characters

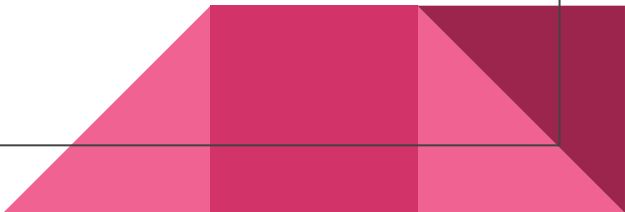
Commands can use wildcards to perform actions on more than one file at a time, or to find part of a phrase in a text file

**Standard Wildcard characters** - used by various command-line utilities to work with multiple files.

man 7 glob

**Regular expressions** - used to manipulate parts of text.

man 7 regex



# Standard Wildcard

- ? (question mark) - Represents any *single* character.
  - Example: "ab?" - Look for aba, abb, abc and ab followed by every other letter/number between a-z, 0-9.
- \* (asterisk) - Represent any number of characters (including zero, in other words, zero or more characters).
  - Example - "ab\*" it would use "aba", "abcde", "abczyta" and *anything* that starts with "ab" also including "ab" itself.
  - "p\*I" could be pill, pull, pl, and anything that starts with an p and ends with an I.
- [] (square brackets) - Specifies a range.
  - Example - If you did r[a,o,i]m it can become: ram, rim, rom
  - If you did: r[a-d]m it can become anything that starts and ends with m and has any character a to d in between.

# Standard Wildcard

- `{ }` (curly brackets) - Terms are separated by commas and each term must be the name of something or a wildcard.
  - Example - `{*.doc,*.pdf}` - Anything ending with `.doc` or `.pdf`. Note that spaces are not allowed after the commas (or anywhere else). `cp {*.doc,*.pdf} ~` (This wildcard will copy anything that matches either wildcard(s), or exact name(s) (an “or” relationship, one or the other).
- `[!]` - This construct is similar to the `[ ]` construct, except rather than matching any characters inside the brackets, it'll match any character, as long as it is not listed between the `[` and `]`. This is a logical NOT.
  - Example `rm myfiles[!9]` will remove all `myfiles*` (ie. `myfiles1`, `myfiles2` etc) but won't remove a file with the number 9 anywhere within it's name.

# Standard Wildcard

- \ (backslash) - Used as an "escape" character, i.e. to protect a subsequent - special character.
  - Example - Thus, "\\" searches for a backslash.





# Regular expressions

- . (dot) - Will match *any single character*, equivalent to ? (question mark) in standard wildcard expressions.
  - Example - "a.c" matches "aac" and "abc" but not "ac" or "abbc".
- \ (backslash) - Used as an "escape" character, i.e. to protect a subsequent special character. Thus, "\\" searches for a backslash.
- \* (asterisk) - The preceding item is to be matched *zero or more* times. ie. n\* will match n, nn, nnnn, nnnnnnnn but not na or any other character.
- .\* (dot and asterisk) - Used to match any string, equivalent to \* in standard wildcards.
- ^ (caret) - The beginning of the line".
  - Example - "^a" means find a line starting with an "a".

# Regular expressions

- **\$ (dollar sign)** - The end of the line".
  - Example - "a\$" means a line ending with an "a".
- **[] (square brackets)** - Specifies a range.
  - Example - If you did `m[a,o,u]m` it can become: mam, mum, mom if you did: `m[a-d]m` it can become anything that starts and ends with m and has any character a to d in between. This kind of wildcard specifies an "or" relationship (you only need one to match).
- **|** - This wildcard makes a logical OR relationship between wildcards. You may need to add a `\` (backslash) before this command to work, because the shell may attempt to interpret this as a pipe.
- **[^]** - Equivalent of `[!]` in standard wildcards. This performs a logical "not". This will match anything that is not listed within those square brackets.
  - Example - `rm myfiles[^9]` will remove all `myfiles*` (ie. `myfiles1`, `myfiles2` etc) but won't remove a file with the number 9 anywhere within it's name.

# Examples

Expression	Matches
<code>tolstoy</code>	The seven letters <code>tolstoy</code> , anywhere on a line
<code>^tolstoy</code>	The seven letters <code>tolstoy</code> , at the beginning of a line
<code>tolstoy\$</code>	The seven letters <code>tolstoy</code> , at the end of a line
<code>^tolstoy\$</code>	A line containing exactly the seven letters <code>tolstoy</code> , and nothing else
<code>[Tt]olstoy</code>	Either the seven letters <code>Tolstoy</code> , or the seven letters <code>tolstoy</code> , anywhere on a line
<code>tol.toy</code>	The three letters <code>tol</code> , any character, and the three letters <code>toy</code> , anywhere on a line
<code>tol.*toy</code>	The three letters <code>tol</code> , any sequence of zero or more characters, and the three letters <code>toy</code> , anywhere on a line (e.g., <code>toltoy</code> , <code>tolstov</code> , <code>tolWHOtoy</code> , and so on)

# Character Categories

- [:upper:] uppercase letters
- [:lower:] lowercase letters
- [:alpha:] alphabetic (letters) meaning upper+lower (both uppercase and lowercase letters)
- [:digit:] numbers in decimal, 0 to 9
- [:alnum:] alphanumeric meaning alpha+digits (any uppercase or lowercase letters or any decimal digits)
- [:space:] whitespace meaning spaces, tabs, newlines and similar




# Character Categories

- `[:punct:]` punctuation characters meaning graphical characters minus alpha and digits
- `[:cntrl:]` control characters meaning non-printable characters
- `[:xdigit:]` characters that are hexadecimal digits.

Example: `ls -l | grep '[:upper:][:digit:]'`

The command greps for `[upper_case_letter][any_digit]`, meaning any uppercase letter followed by any digit




# Commands

# Grep

Search a file for a pattern

Some useful options:

- -F : Match using fixed strings.
  - -e pattern\_list : Specify one or more patterns to be used during the search for input.
  - -f pattern\_file : Search from a file
  - -i : Perform pattern matching in searches without regard to case
- 

# Grep

- **-n** Precede each output line by its relative line number in the file, each file starting at line 1.
- **-q** No output to stdout
- **-v** Select lines not matching any of the specified patterns.
- **-x** Consider only input lines that use all characters to match an entire fixed string or regular expression to be matching lines.

The standard input shall be used if no *file* operands are specified



# Grep

```
grep -E '^Bat' myfile.txt
```

```
grep -E 'man$' myfile.txt
```

```
grep -E '^(bat|Bat|cat|Cat)' myfile.txt
```

```
grep -i -E '^(bat|cat)' myfile.txt
```

```
grep -E '^[bcBC]at' myfile.txt
```

```
grep -i -E '^[^b]at' myfile.txt
```



# Sed

**Sed** is a stream editor. Used to perform basic text transformations on an input stream (a file or input from a pipeline).

```
sed 's/regExpr/replText/[g]'
```



# Sed Examples

- `sed 's/Nick/John/g' report.txt` - Replace every occurrence of Nick with John in report.txt
- `sed 's/[N|n]ick/John/g' report.txt` - Replace every occurrence of Nick or nick with John.
- `sed '$d' file.txt` - Delete the last line
- `sed '/[0-9]\{3\}/p' file.txt` - Print only lines with three consecutive digits
- `sed '17,/disk/d' file.txt` - Delete all lines from line 17 to 'disk'



# Other important commands

wc: outputs a one-line report of lines, words, and bytes

head: extract top of files

tail: extracts bottom of files

tr: translate or delete characters

sort: sort lines of text files

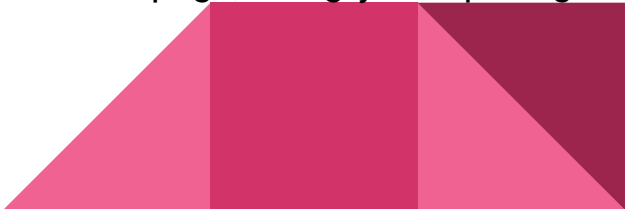


# LAB 2

# Introduction

Build a spelling checker for the Hawaiian language  
(Get familiar with sort, comm and tr commands!)

Steps:

- Download a copy of web page containing basic English-to-Hawaiian dictionary
  - Extract only the Hawaiian words from the web page to build a simple Hawaiian dictionary. Save it to a file called hwords (site scraping)
  - Automate site scraping: buildwords script (`cat file.html | buildwords > hwords`)
  - Modify the command in the lab assignment to act as a spelling checker for Hawaiian
  - Use your spelling checker to check hwords and the lab web page for spelling mistakes
  - Count the number of "misspelled" English and Hawaiian words on this web page, using your spelling checkers.
- 

# Hints

- Run your script on seasnet servers before submitting to CCLE
- `sed '/patternstart/,/patternstop/d'`
  - delete patternstart to patternstop, works across multiple lines. Will delete all lines starting with patternstart to patternstop
- The Hawaiian words html page uses `\r` and `\n` for new lines
  - `od -c hwnwdseng.htm` to see the ASCII characters
- You can delete blank white spaces such as tab or space using
  - `tr -d '[:blank:]'`
- Use `tr -s` to squeeze multiple new lines into one
- `sed 's/<[^>]*>//g' a.html` to remove all HTML tags

# References

<http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/x11655.htm>

<https://www.ibm.com/developerworks/aix/library/au-speakingunix9/>

<https://linuxconfig.org/learning-linux-commands-sed>





# CS 35L

LAB 8, Session 3

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

- Basics of Python
- Java as a compromise between compiled and interpreted languages
- Lab assignment



# Details

`mkdir lab3_1` (Make a directory for each lab session)

`cd lab3_1`

`touch lab.log` (optional)

`touch lab.txt`

`touch hw.txt`





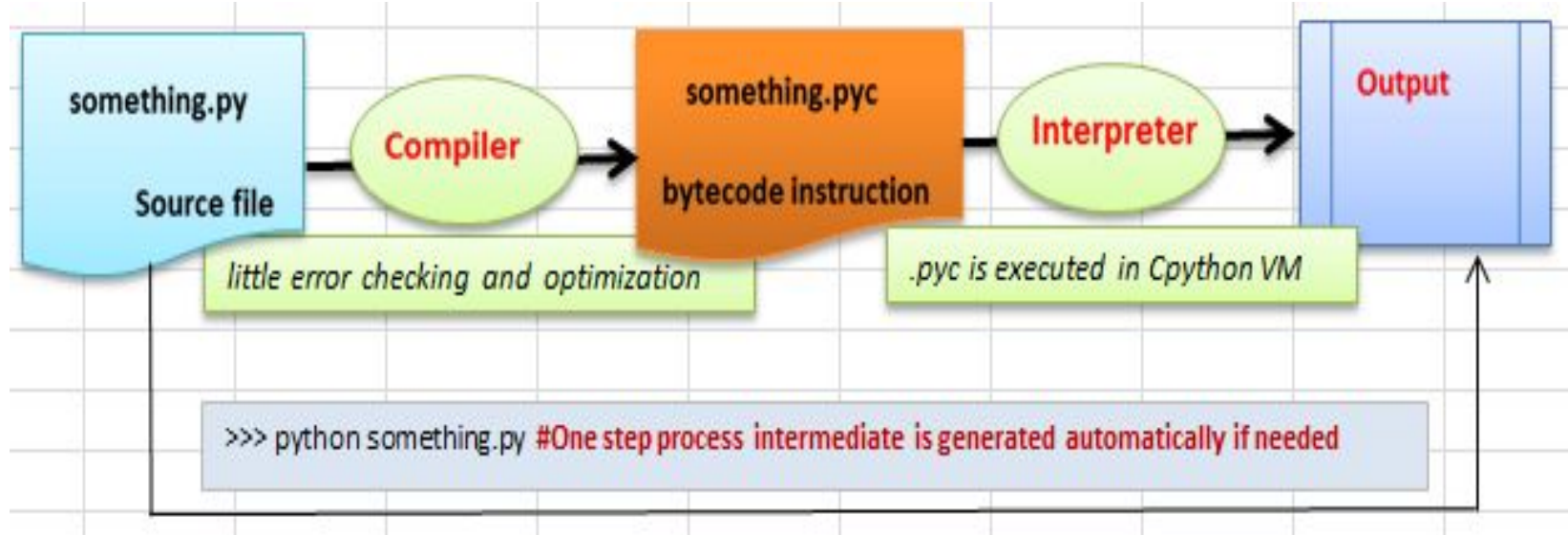
# PYTHON

# Introduction

- High level programming language
- General purpose
- Byte Code Interpreted
- Dynamic programming language
- Object Oriented



# Introduction



Byte code can be interpreted (official CPython), or JIT compiled (PyPy)

# Hello world

Python

```
print "Hello\nWorld"
```

```
print 'Hello\nWorld'
```



# Math Operations

- Addition
- Subtraction
- Multiplication
- Division (22/3, 22/3.0)
- Remainder
- Exponent
- Substitution
  - `print "1+2 is"`
  - `print "", 1+2, "is"`





# Constructs

- Variables
- String manipulation
- For loops
- If
- If else
- If elif else



# For loops

```
for iterating_var in sequence:  
    statements(s)
```




# Functions

```
def functionname( parameters ):  
    "function_docstring"  
    function_suite  
    return [expression]
```



# Functions

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```




# Lists, Tuples, Dictionary

Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Dictionary is essentially a hash table i.e a key value pair. Keys are unique, values are not



# Class

```
class ClassName:
```

```
    'Optional class documentation string'
```

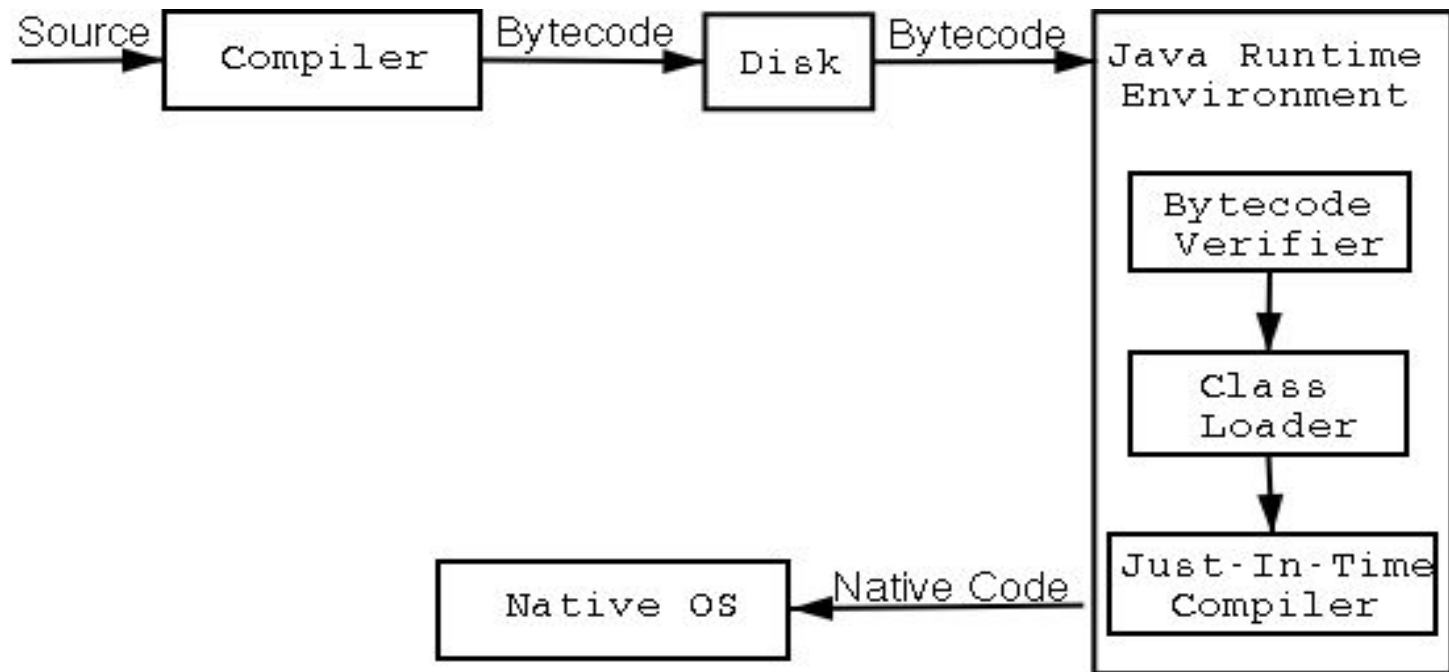
```
    class_suite
```





# JAVA

# Java





# CS 35L

LAB 8,

TA: Sucharitha Prabhakar

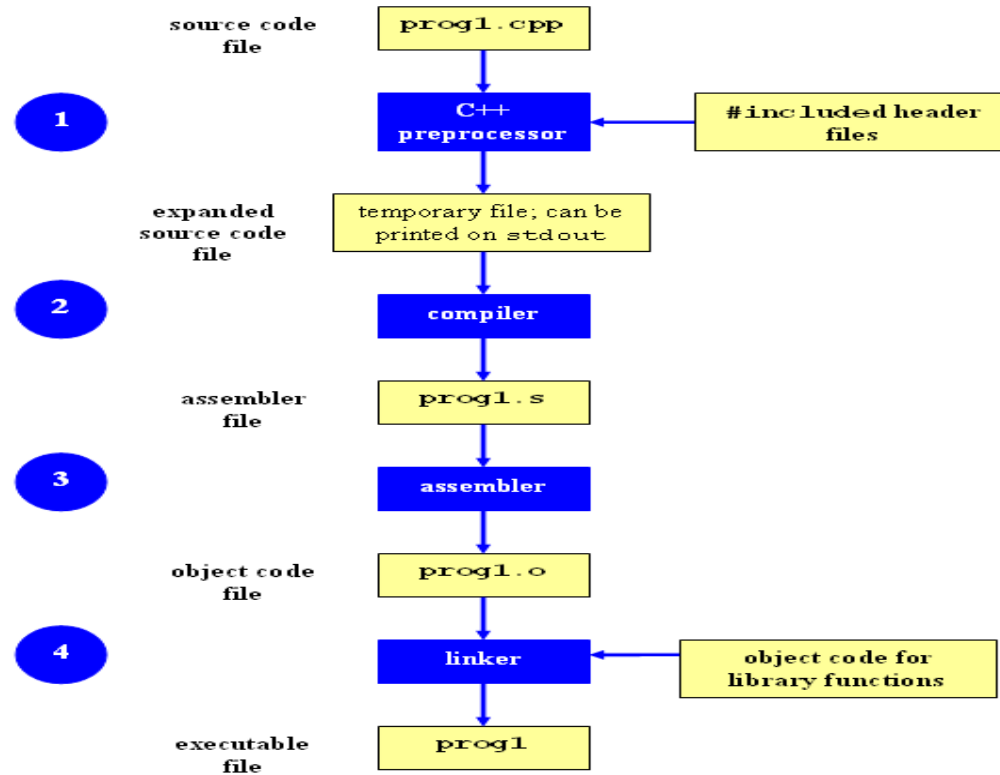
EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

- Compilation process
- Make
- Automake
- Applying a patch



# Compilation process



# Compilation process

## Preprocessing:

```
cc -E hello_world.c
```

## Compilation:

```
cc -S hello_world.c
```

## Assembly:

```
cc -c hello_world.c
```

```
hexdump hello_world.o
```

## Linking

```
cc -o hello_world hello_world.c
```



# Compilation

- shop.cpp
  - #includes shoppingList.h and item.h
- shoppingList.cpp
  - #includes shoppingList.h
- item.cpp
  - #includes item.h
- How to compile?
  - `g++ -Wall shoppingList.cpp item.cpp shop.cpp -o shop`



# What if ?

- We change one of the header or source files?
  - Rerun command to generate new executable
- We only made a small change to item.cpp?
  - not efficient to recompile shoppinglist.cpp and shop.cpp
- Solution: avoid waste by producing a separate object code file for each source file
  - `g++ -Wall -c item.cpp...` (for each source file)
  - `g++ item.o shoppingList.o shop.o -o shop` (combine)
- Less work for compiler, saves time but more commands

# What if ?

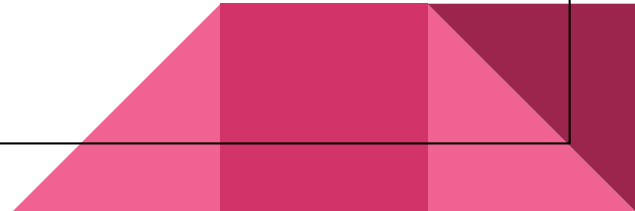
- We change item.h?
- Need to recompile every source file that includes it & every source file that includes a header that includes it. Here: item.cpp and shop.cpp
- Difficult to keep track of files when project is large
  - Windows 7 ~40 million lines of code
  - Google ~2 billion lines of code
- **MAKE**



# Make

GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.

Make gets its knowledge of how to build your program from a file called the makefile, which lists each of the non-source files and how to compute it from other files.





# Advantages of Make

Make enables the end user to build and install your package without knowing the details of how that is done

Make figures out automatically which files it needs to update, based on which source files have changed. It also automatically determines the proper order for updating files, in case one non-source file depends on another non-source file.

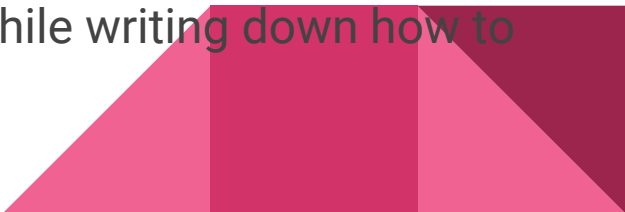
As a result, if you change a few source files and then run Make, it does not need to recompile all of your program.



# Advantages of Make

Make is not limited to any particular language. For each non-source file in the program, the makefile specifies the shell commands to compute it. These shell commands can run a compiler to produce an object file, the linker to produce an executable, or to update a library, or TeX or Makeinfo to format documentation.

Make is not limited to building a package. You can also use Make to control installing or uninstalling a package, generate tags tables for it, or anything else you want to do often enough to make it worth while writing down how to do it.



- Configure

## Build Process

- Script that checks details about the machine before installation

- Dependency between packages

- Creates 'Makefile'

- make

- Requires 'Makefile' to run

- Compiles all the program code and creates executables in current temporary directory

- make install

- make utility searches for a label named install within the Makefile, and executes only that section of it

- executables are copied into the final directories (system directories)


# Automake

**GNU Automake** is a tool to automate parts of the compilation process.

It eases usual compilation problems. For example, it points to needed dependencies.

It automatically generates one or more *Makefile.in* from files called *Makefile.am*.

Each *Makefile.am* contains, among other things, useful variable definitions for the compiled software, such as compiler and linker flags, dependencies and their versions, etc.




# Automake

**GNU Automake** is a tool to automate parts of the compilation process.

It eases usual compilation problems. For example, it points to needed dependencies.

It automatically generates one or more *Makefile.in* from files called *Makefile.am*.

Each *Makefile.am* contains, among other things, useful variable definitions for the compiled software, such as compiler and linker flags, dependencies and their versions, etc.



# Automake

Makefile.am in <tests/project>

```
SUBDIRS = src
```

Makefile.am in <tests/project/src>

```
bin_PROGRAMS = helloworld
```

```
AM_CXXFLAGS = $(INTI_CFLAGS)
```

```
helloworld_SOURCES = main.cc helloworld.cc helloworld.h
```

```
helloworld_LDADD = $(INTI_LIBS)
```



# Lab Assignment

# Installing Software

- Windows

- Installshield
- Microsoft/Windows Installer

- OS X

- Drag and drop from .dmg mount -> Applications folder

- Linux

- rpm(Redhat Package Management)
- RedHat Linux (.rpm)





# Decompressing files

- Generally, you receive Linux software in the tarball format (.tgz) or (.gz)
- Decompress file in current directory:
  - `$ tar -xzvf filename.tar.gz`
  - Option `-x`: --extract
  - Option `-z`: --gzip
  - Option `-v`: --verbose
  - Option `-f`: --file



# Problem

- Coreutils 7.6 has a problem
  - Different users see different date formats
  - `$ ls -l /bin/bash`
    - `-rwxr-xr-x 1 root root 729040 2009-03-02 06:22 /bin/bash`
    - `-rwxr-xr-x 1 root root 729040 Mar 2 2009 /bin/bash`
  - Why?
    - Different locales
  - Want the traditional Unix format for all users
  - Fix the ls program




# Setup

- Download coreutils-7.6 to your home directory
  - Use 'wget'
- Untar and Unzip it
  - `tar -xzf coreutils-7.6.tar.gz`
- Make a directory `~/coreutilsInstall` in your home directory (this is where you'll be installing coreutils)
  - `mkdir ~/coreutilsInstall`



# Setup

- Go into coreutils-7.6 directory. This is what you just unzipped.
  - Read the INSTALL file on how to configure “make”, especially --prefix flag
  - Run the configure script using the prefix flag so that when everything is done, coreutils will be installed in the directory ~/coreutilsInstall
  - Compile it: make
  - Install it: make install
- 

# Reproduce bug

Reproduce the bug by running the version of 'ls' in coreutils 7.6

If you just type `ls` at CLI it won't run 'ls' in coreutils 7.6

Why? Shell looks for `/bin/ls`

To use coreutils 7.6: `$ ./ls -lrt`

This manually runs the executable in this directory (ls is in src directory)

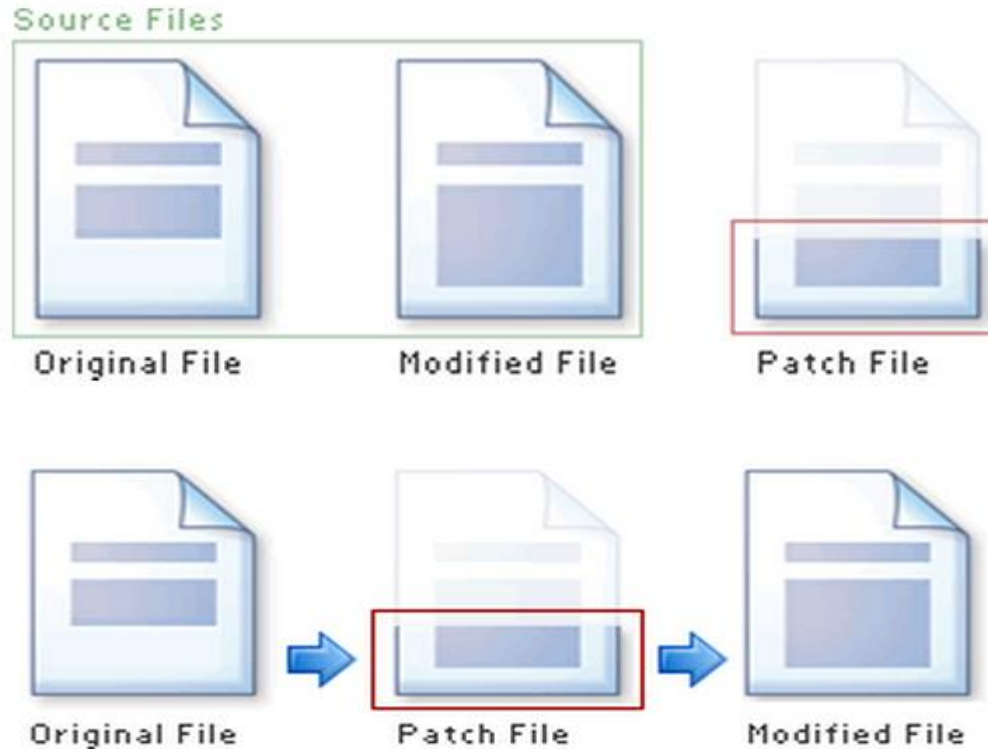


# Patch

- A patch is a piece of software designed to fix problems with or update a computer program
- It's a diff file that includes the changes made to a file
- A person who has the original (buggy) file can use the patch command with the diff file to add the changes to their original file



# Applying a Patch



# diff Unified Format

```
diff -u original_file modified_file
```

```
--- path/to/original_file
```

```
+++ path/to/modified_file
```

```
@@ -l,s +l,s @@
```

@@: beginning of a chunk

l: beginning line number

s: number of lines the change chunk applies to for each file

A line with a:



# Patching and Building

`cd coreutils-7.6`

`vim` or `emacs` `patch_file`: copy and paste the patch content

`patch -pnum < patch_file`

‘`man patch`’ to find out what `pnum` does and how to use it

`cd` into the `coreutils-7.6` directory and type `make` to rebuild patched `ls`

Don't install!!



# Testing fix

- Test the following:
  - Modified ls works
  - Installed unmodified ls does NOT work
- Test on:
  - A file that has been recently modified
    - Make a change to an existing file or create a new file
  - A file that is at least a year old
    - `touch -t 201401210959.30 test_file`



# CS 35L

LAB 8,

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

Introduction to C



# Basic data types

int

Holds integer numbers

Usually 4 bytes

float

Holds floating point numbers

Usually 4 bytes

double

Holds higher-precision floating point numbers

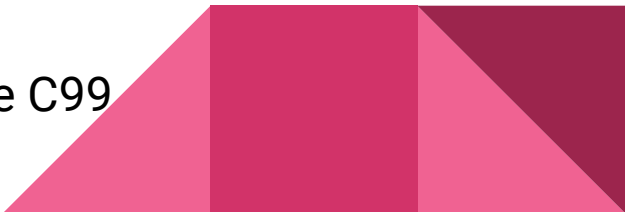
Usually 8 bytes (double the size of a float)

char

Holds a byte of data, characters

Void

Pretty much like C++ basic data types, but NO bool before C99



# Pointers

Variables that store memory addresses

Declaration

**<variable\_type> \*<name>;**

```
int *ptr;    //declare ptr as a pointer to int
```

```
int var = 77;    // define an int variable
```

```
ptr = &var;    // let ptr point to the variable var
```



# Dereferencing Pointers

Accessing the value that the pointer points to

Example:

```
double x, *ptr;
```

```
ptr = &x;    // let ptr point to x
```

```
*ptr = 7.8;  // assign the value 7.8 to x
```

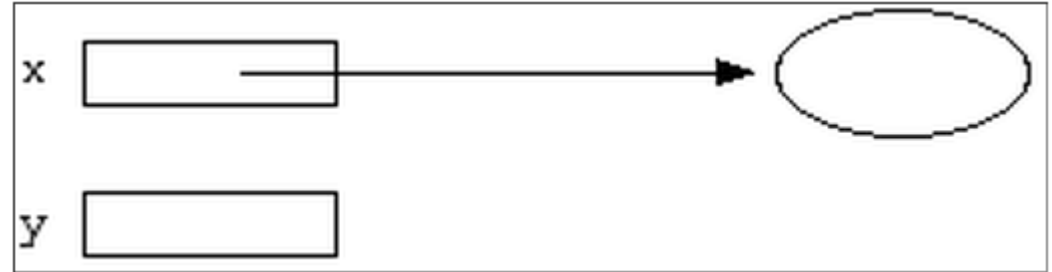


# Pointer Example

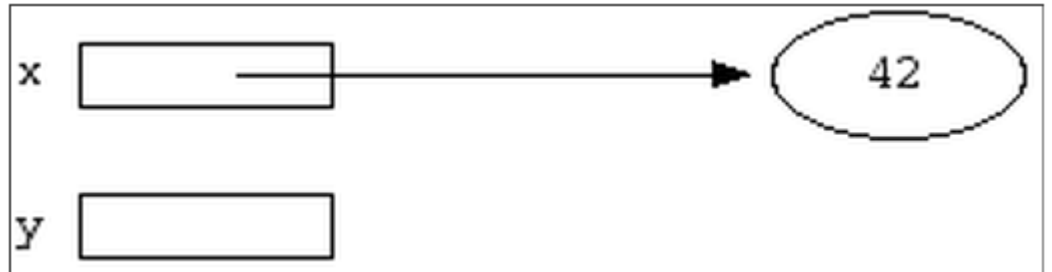
```
int *x; int *y;
```



```
int var; x = &var;
```



```
*x = 42;
```





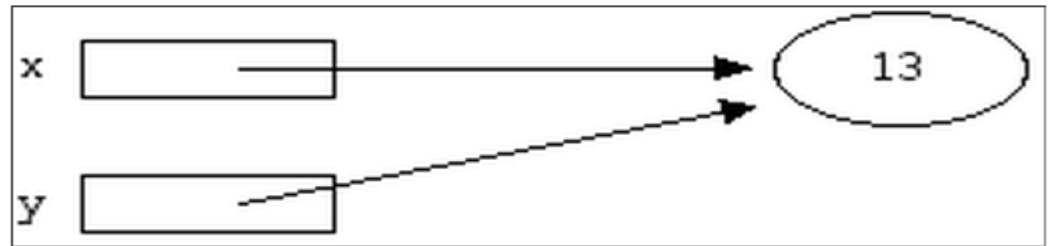
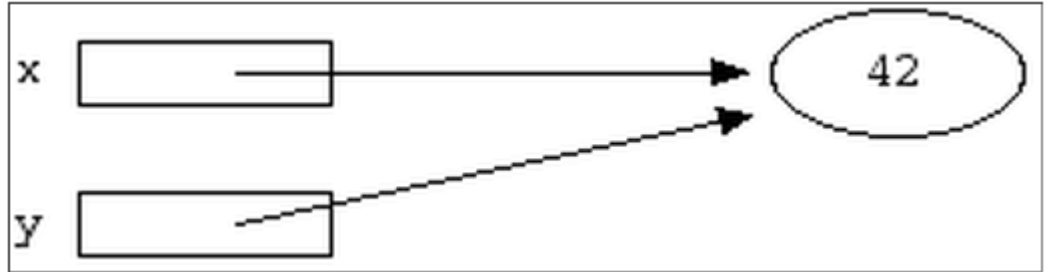
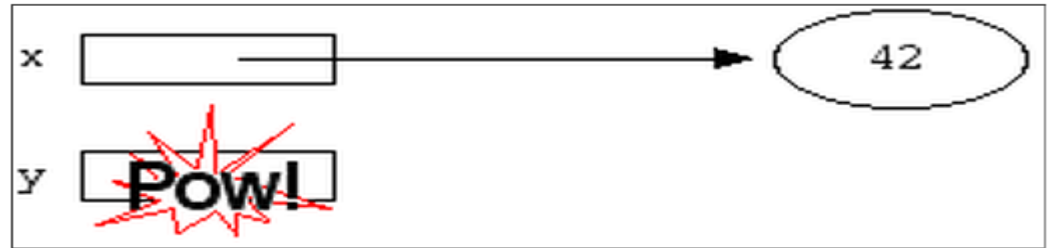
# Pointer Example

`*y = 13;`

`y = x;`

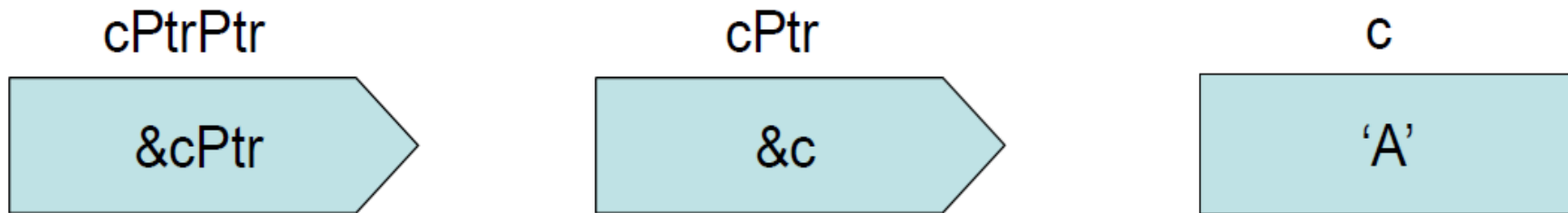
`*x = 13; or`

`*y = 13;`



# Pointers to Pointers

```
char c = 'A' ;      *cPtr = &c;      **cPtrPtr = &cPtr;
```



# Pointers to Functions

Also known as: function pointers or functors

Declaration: <function return type>(\*<Pointer\_name>)(function argument list)

Example: double (\*p2f)(double, char, int)



# Pointers to Functions - Example

**Goal: write a sorting function**

**Has to work for ascending and descending sorting order**

How?

Write multiple functions

Provide a flag as an argument to the function

Use function pointers!!



# Pointers to Functions - Example

User can pass in a function to the sort function

## Declaration

```
double (*func_ptr) (double, double);
```

```
func_ptr = pow; // func_ptr points to pow()
```

## Usage

```
// Call the function referenced by func_ptr
```

```
double result = (*func_ptr)( 1.5, 2.0 );
```

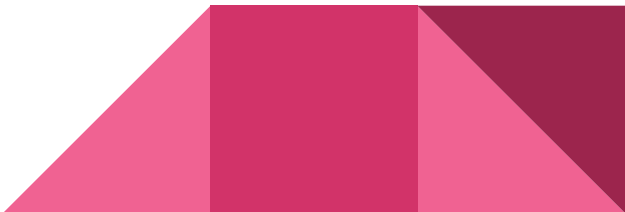
```
// The same function call
```

```
double result = func_ptr( 1.5, 2.0 );
```



# Pointers to Functions - qsort

```
#include <stdio.h>
#include <stdlib.h>
int compare (const void * a, const void * b) {
    return ( *(int*)a - *(int*)b );
}
int main () {
    int values[] = { 40, 10, 100, 90, 20, 25 };
    qsort (values, 6, sizeof(int), compare);
    int n;
    for (n = 0; n < 6; n++) {
        printf ("%d ",values[n]);
    }
    return 0;
}
```



# Structs

No classes in C

Used to package related data (variables of different types) together

Single name is convenient

```
struct Student {  
    char name[64];  
    char UID[10];  
    int age;  
    int year;  
};  
struct Student s;
```

```
typedef struct {  
    char name[64];  
    char UID[10];  
    int age;  
    int year;  
} Student;  
Student s;
```

# C structs vs. C++ classes

C structs cannot have member functions

C++ classes can have member functions

There's no such thing as access specifiers in C

C++ class members have access specifiers and are private by default

C structs don't have constructors defined for them

C++ classes must have at least a default constructor



# Dynamic Memory

Memory that is allocated at runtime

Allocated on the heap

**void \*malloc (size\_t size);**

Allocates size bytes and returns a pointer to the allocated memory

**void \*realloc (void \*ptr, size\_t size);**

Changes the size of the memory block pointed to by ptr to size bytes

**void free (void \*ptr);**

Frees the block of memory pointed to by ptr



# Reading and Writing Chars

**int getchar();**

Returns the next character from stdin

**int putchar(int character);**

Writes a character to the current position in stdout



# Formatted I/O

```
int fprintf(FILE * fp, const char * format, ...);
```

```
int fscanf(FILE * fp, const char * format, ...);
```

FILE \*fp can be either:

- A file pointer

- stdin, stdout, or stderr

The format string

```
int score = 120; char player[] = "Mary";
```

```
printf("%s has %d points.\n", player, score);
```



# Homework 5

Write a C program called sfrob

Reads stdin byte-by-byte (getchar)

Consists of records that are space-delimited

Each byte is frobnicated (XOR with dec 42)

Sort records without decoding (qsort, frobcmp)

Output result in frobnicated encoding to stdout (putchar)

Error checking (fprintf)

Dynamic memory allocation (malloc, realloc, free)



# Homework 5

Input: `printf 'sybjre obl'`

`$ printf 'sybjre obl ' | ./sfrob`

Read the records: `sybjre, obl`

Compare records using `frobcmp` function

Use `frobcmp` as compare function in `qsort`

Output: `obl sybjre`



# CS 35L

LAB 8,

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

**Processor modes**

**System calls**



# Processor Modes/ CPU modes

Operating modes that place restrictions on the type of operations that can be performed by running processes.

- User mode: restricted access to system resources
- Kernel/Supervisor mode: unrestricted access





# User mode vs Kernel mode

Hardware contains a mode-bit, e.g. 0 means kernel mode, 1 means user mode

- User mode
  - CPU restricted to unprivileged instructions and a specified area of memory
- Supervisor/kernel mode
  - CPU is unrestricted, can use all instructions, access all areas of memory and take over the CPU anytime



# Why dual mode?

System resources are shared among processes

OS must ensure:

- Protection
  - An incorrect/malicious program cannot cause damage to other processes or the system as a whole
- Fairness
  - Make sure processes have a fair use of devices and the CPU

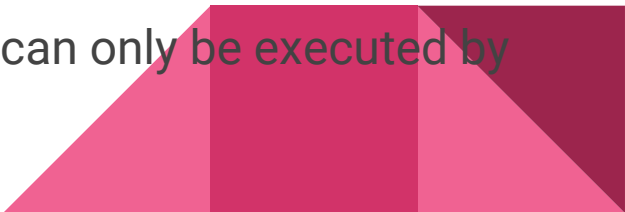


# Goals for protection

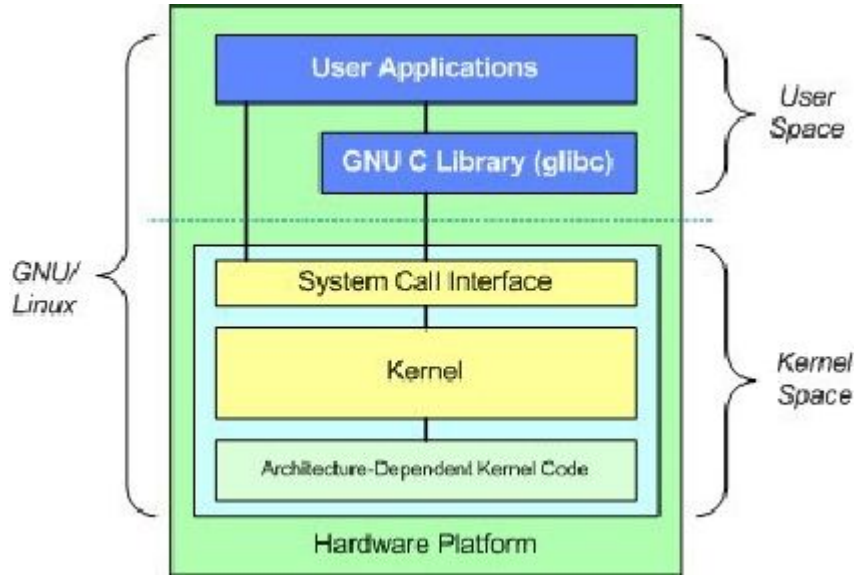
Goals:

- I/O Protection
  - Prevent processes from performing illegal I/O operations
- Memory Protection
  - Prevent processes from accessing illegal memory and modifying kernel code and data structures
- CPU Protection
  - Prevent a process from using the CPU for too long

=> instructions that might affect goals are privileged and can only be executed by trusted code



# Why code is trusted only in Kernel mode?

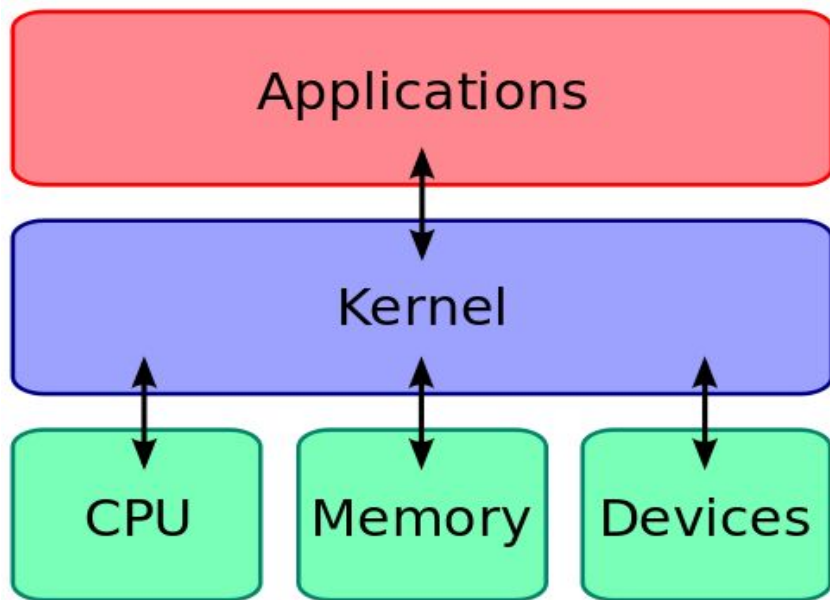


Core of OS software executing in supervisor state

Trusted software:

- Manages hardware resources (CPU, Memory and I/O)
- Implements protection mechanisms that could not be changed through actions of untrusted software in user space
- System call interface is a safe way to expose privileged functionality and services of the processor

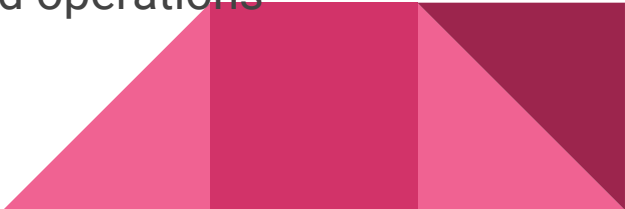
# What happens to user processes?



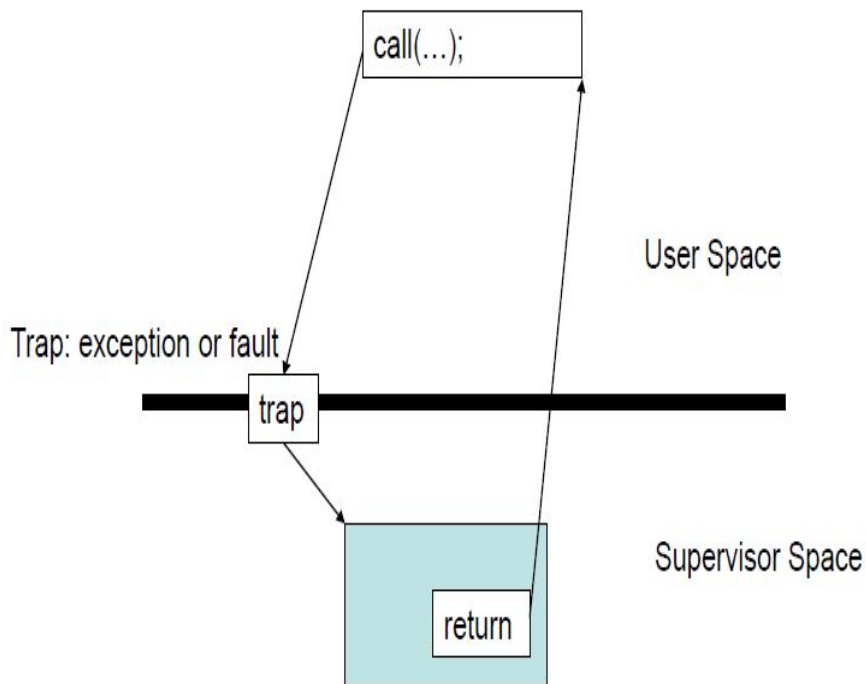
The kernel executes privileged operations on behalf of untrusted user processes

# System calls

Special type of function that:

- Used by user-level processes to request a service from the kernel
  - Changes the CPU's mode from user mode to kernel mode to enable more capabilities
  - Is part of the kernel of the OS
  - Verifies that the user should be allowed to do the requested action and then does the action (kernel performs the operation on behalf of the user)
  - Is the only way a user program can perform privileged operations
- 

# System calls



When a system call is made, the program being executed is interrupted and control is passed to the kernel

If operation is valid the kernel performs it

# System Call Overhead

System calls are expensive and can hurt performance

The system must do many things

- Process is interrupted & computer saves its state
- OS takes control of CPU & verifies validity of op.
- OS performs requested action
- OS restores saved context, switches to user mode
- OS gives control of the CPU back to user process





# Example system calls

**READ** - To access data from a file stored in a file system use the **read** system call. This system call reads in data in bytes, the number of which is specified by the caller, from the file and stores then into a buffer supplied by the calling process

```
ssize_t read(int fd, void *buf, size_t count);
```

**WRITE** - It writes data from a buffer declared by the user to a given device, maybe a file. This is primary way to output data from a program by directly using a system call.

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```



# Example system calls

OPEN - a program initializes access to a file in a filesystem using the **open** system call. This allocates resources associated to the file and returns a handle that the process will use to refer to that file.

```
FILE *fopen(const char *restrict filename, const char *restrict mode);
```

CLOSE - A program terminates access to a file in a file system using the close system call.

```
int fclose(FILE *stream);
```



# Example system calls

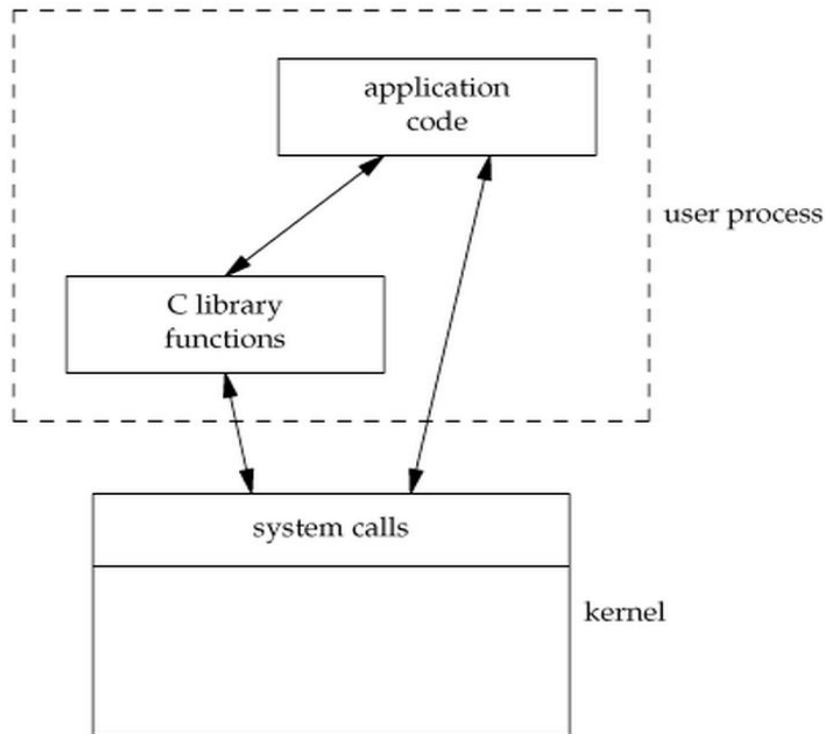
**Fork** - In the context of the UNIX operating system, fork is an operation whereby a process creates a copy of itself.

**Exec** - A functionality of an Operating System that runs an executable file in the context of an already existing process, replacing the previous executable.

**Wait** - A process may wait on another process to complete its execution.



# Point?



Many library functions invoke system calls indirectly

So why use library calls?

- Usually equivalent library functions make fewer system calls
- non-frequent switches from user mode to kernel mode => less overhead

# Unbuffered vs Buffered I/O

- Unbuffered
  - Every byte is read/written by the kernel through a system call
- Buffered
  - Collect as many bytes as possible (in a buffer) and read more than a single byte (into buffer) at a time and use one system call for a block of bytes

=> Buffered I/O decreases the number of read/write system calls and the corresponding overhead



# Lab

Write `tr2b` and `tr2u` programs in 'C' that transliterates bytes. They take two arguments 'from' and 'to'. The programs will transliterate every byte in 'from' to corresponding byte in 'to'

- `./tr2b 'abcd' 'wxyz' < bigfile.txt`
- Replace 'a' with 'w', 'b' with 'x', etc
- `./tr2b 'mno' 'pqr' < bigfile.txt`



# Lab

tr2b uses `getchar` and `putchar` to read from STDIN and write to STDOUT.

tr2u uses `read` and `write` to read and write each byte, instead of using `getchar` and `putchar`. The `nbyte` argument should be 1 so it reads/writes a single byte at a time.

Test it on a big file with 5000000 bytes

```
$ head --bytes=# /dev/urandom > output.txt
```

Example `head --bytes=400 /dev/urandom`



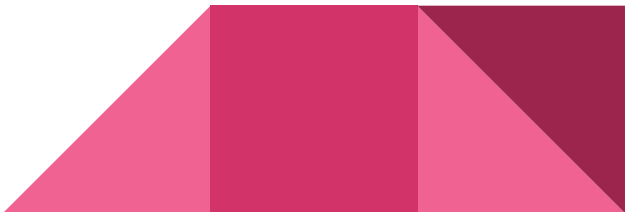
# Hint:

`time [options] command [arguments...]`

Output:

- `real 0m4.866s`: elapsed time as read from a wall clock
- `user 0m0.001s`: the CPU time used by your process
- `sys 0m0.021s`: the CPU time used by the system on behalf of your process

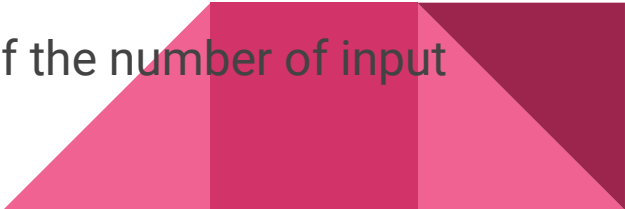
`strace`: intercepts and prints out system calls to `stderr` or to an output file

- `$ strace -o strace_output ./tr2b 'AB' 'XY' < input.txt`
  - `$ strace -o strace_output2 ./tr2u 'AB' 'XY' < input.txt`
- 



# HW

Rewrite sfrob using system calls (sfrobu)

- sfrobu should behave like sfrob except:
  - If stdin is a regular file, it should initially allocate enough memory to hold all data in the file all at once
  - It outputs a line with the number of comparisons performed
  - Functions you'll need: read, write, and fstat (read the man pages)
  - Measure differences in performance between sfrob and sfrobu using the time command
  - Estimate the number of comparisons as a function of the number of input lines provided to sfrobu
- 

# HW

Write a shell script “sfrobs” that uses tr and the sort utility to perform the same overall operation as sfrob

Encrypted input -> tr (decrypt) -> sort (sort decrypted text) -> tr (encrypt) -> encrypted output



# CS 35L

LAB 8,

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

**Multiprocessing**

**Parallelism**

**Multithreading**

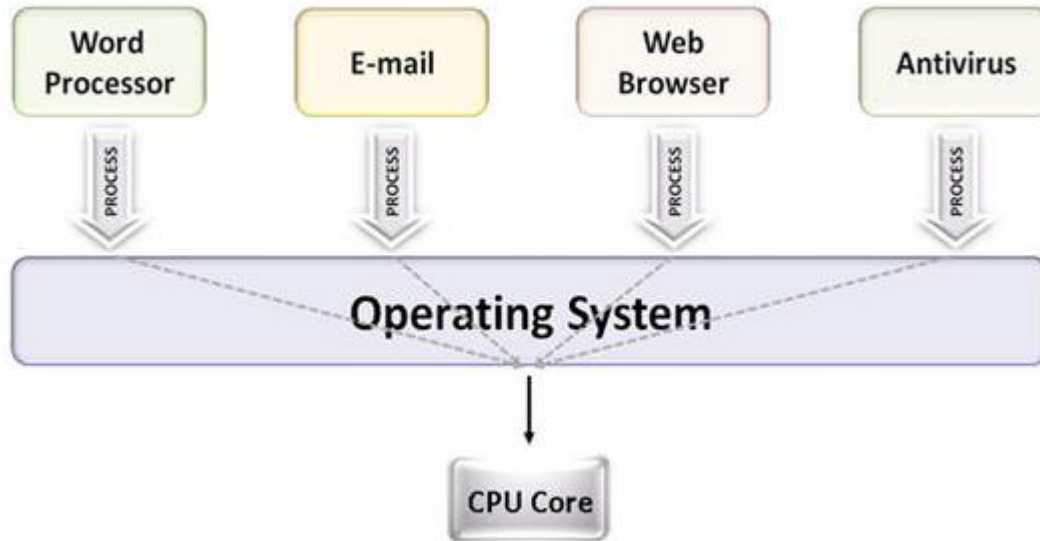


# Multiprocessing

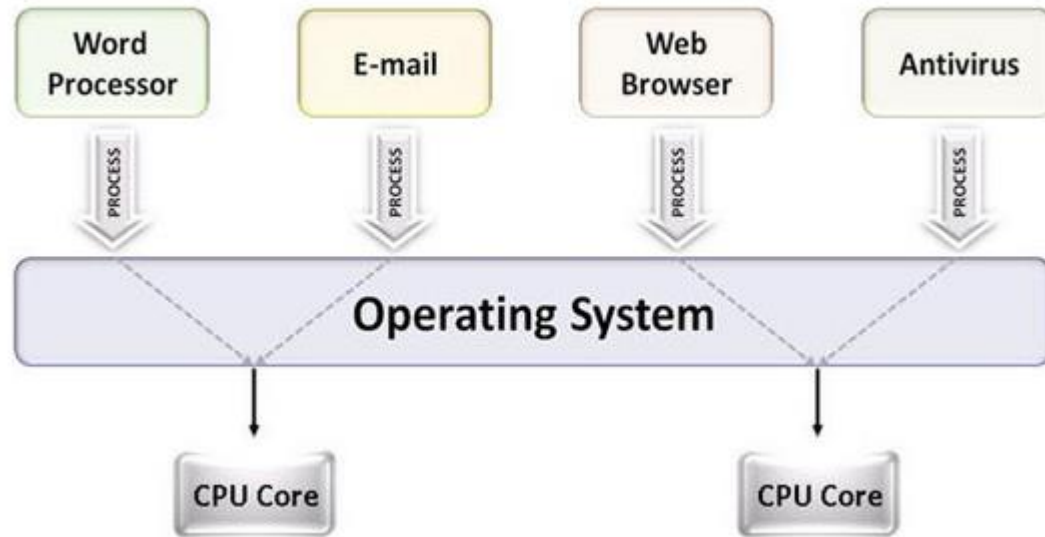
The use of multiple CPUs/cores to run multiple tasks simultaneously



# Uniprocessing



# Multiprocessing



# Parallelism

Executing several computations simultaneously to gain performance

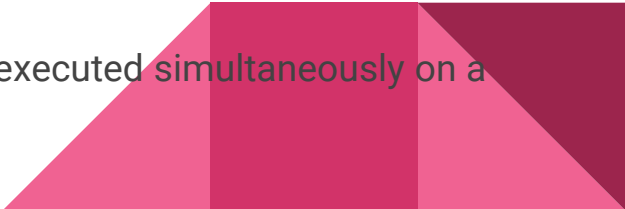
Different forms of parallelism

## Multitasking

Several processes are scheduled alternately or possibly simultaneously on a multiprocessing system

## Multithreading

Same job is broken logically into pieces (threads) which may be executed simultaneously on a multiprocessing system





# What is a thread?

A flow of instructions, path of execution within a process. The smallest unit of processing scheduled by OS. A process consists of at least one thread

Multiple threads can be run on:

A uniprocessor (time-sharing)

Processor switches between different threads

Parallelism is an illusion

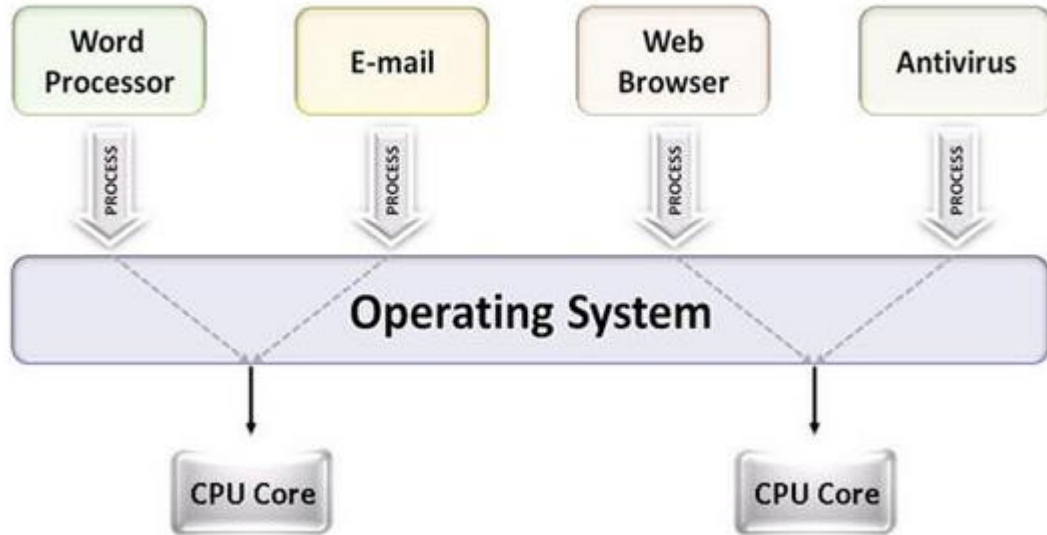
A multiprocessor

Multiple processors or cores run the threads at the same time

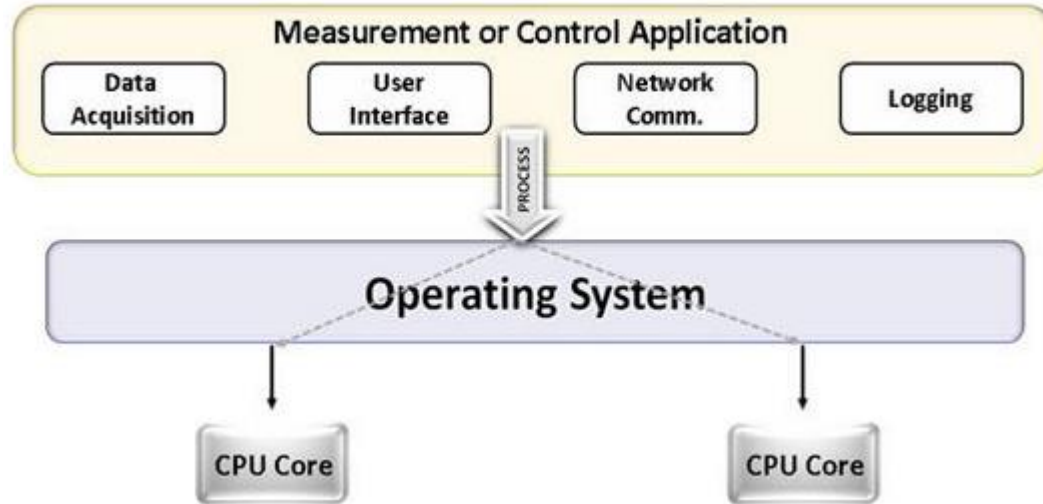
True parallelism



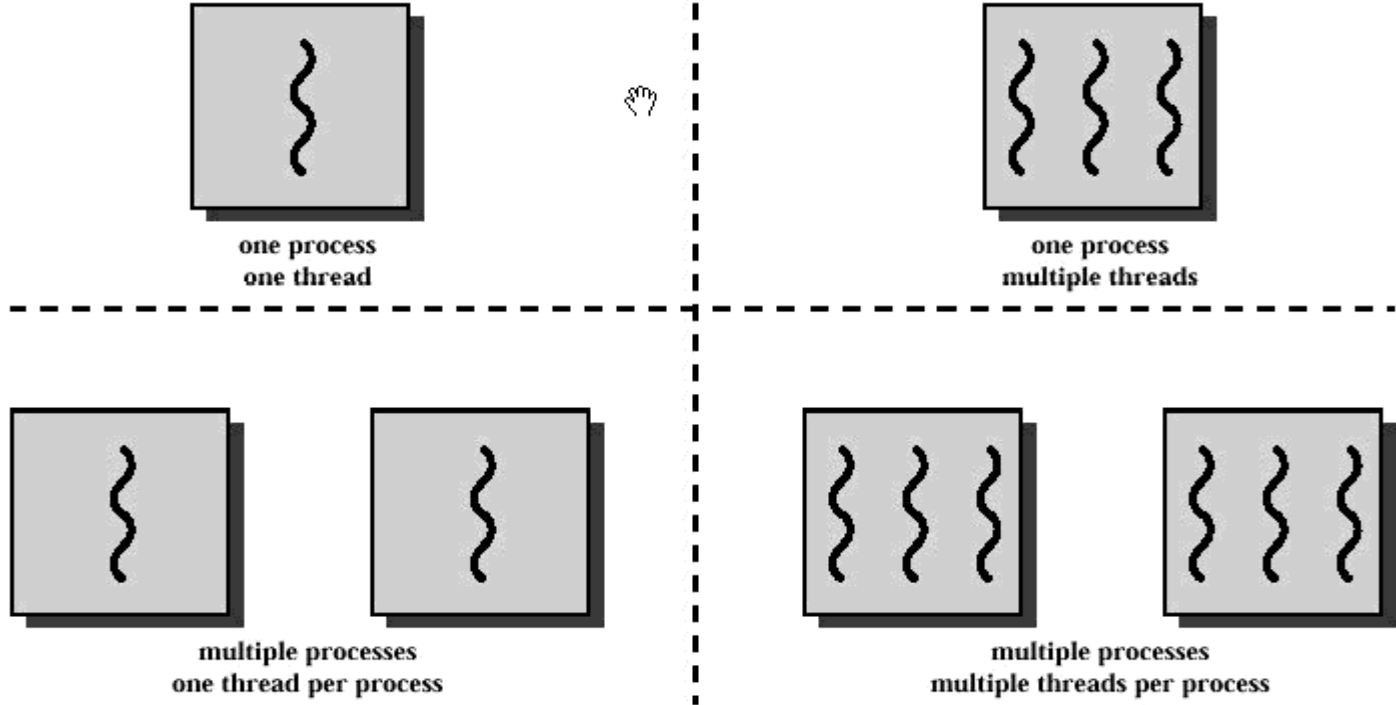
# Multitasking



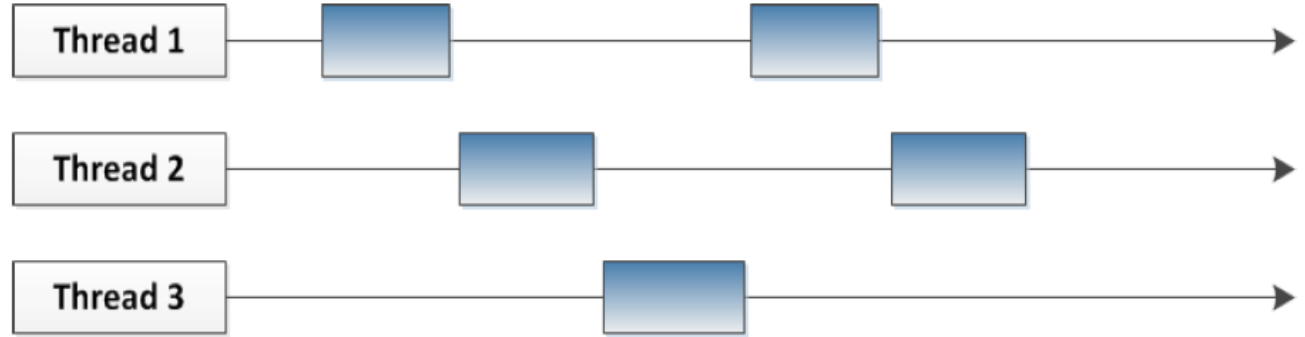
# Multithreading



# Multitasking and Multithreading



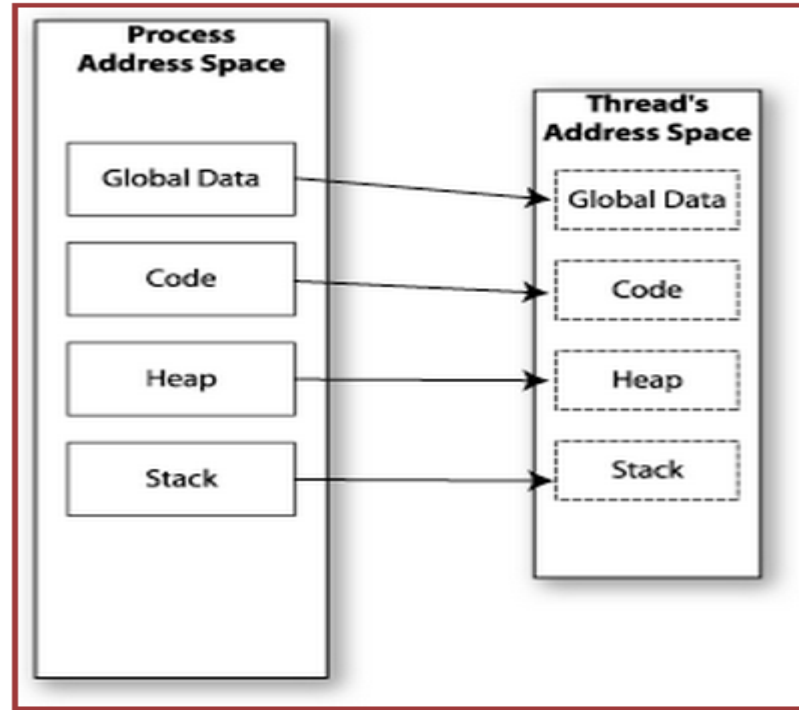
Multiple  
threads  
sharing a  
single CPU



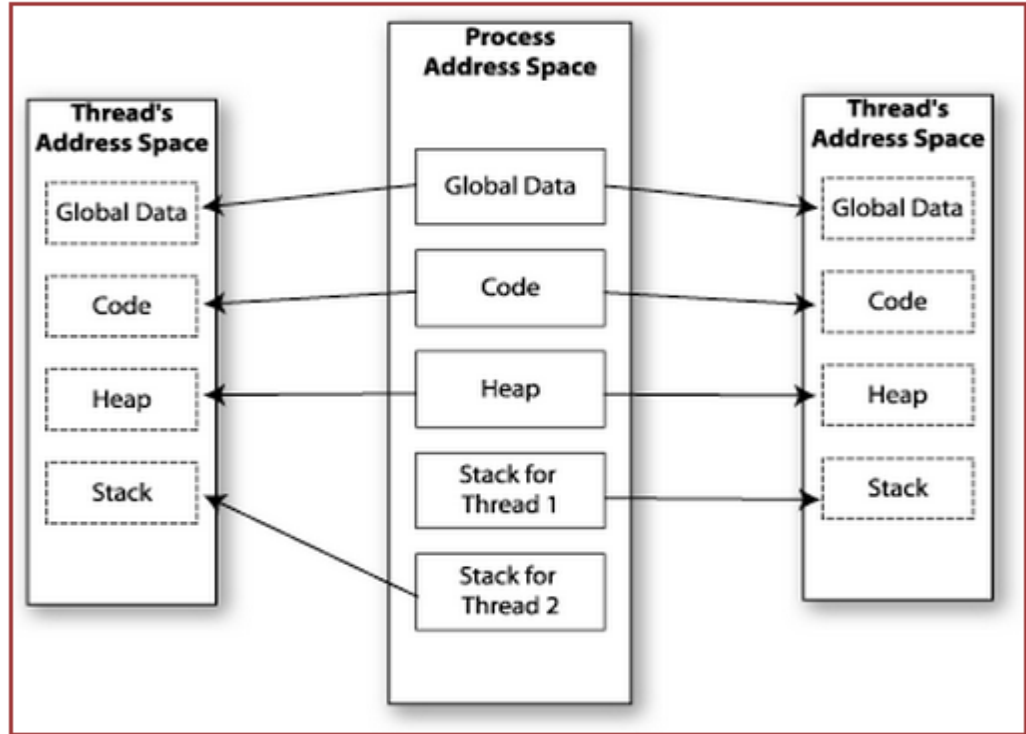
Multiple  
threads on  
multiple  
CPUs



# Memory Layout: Single-Threaded Program



# Memory Layout: Multithreaded Program



Multithreading

# Multithreading & Multitasking: Comparison

Threads share the same address space

Light-weight creation/destruction

Easy inter-thread communication

An error in one thread can bring down all threads in process

Multitasking

Processes are insulated from each other

Expensive creation/destruction

Expensive IPC

An error in one process cannot bring down another process





# Communication in Multitasking

```
tr -cs 'A-Za-z' '[\n*]' | sort -u | comm -23 - words
```

Process 1 (tr), Process 2 (sort), Process 3 (comm)

Each process has its own address space

How do these processes communicate?

Pipes/System Calls



# Communication in Multithreading

Threads share all of the process's memory except for their stacks

=> Data sharing requires no extra work (no system calls, pipes, etc.)



# Shared memory

Makes multithreaded programming

## Powerful

Can easily access data and share it among threads

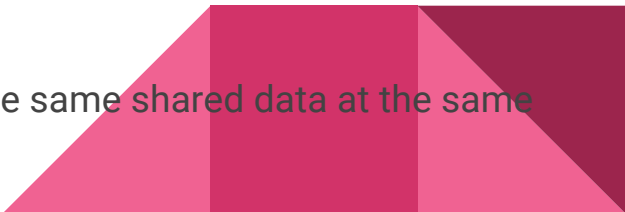
## More efficient

No need for system calls when sharing data

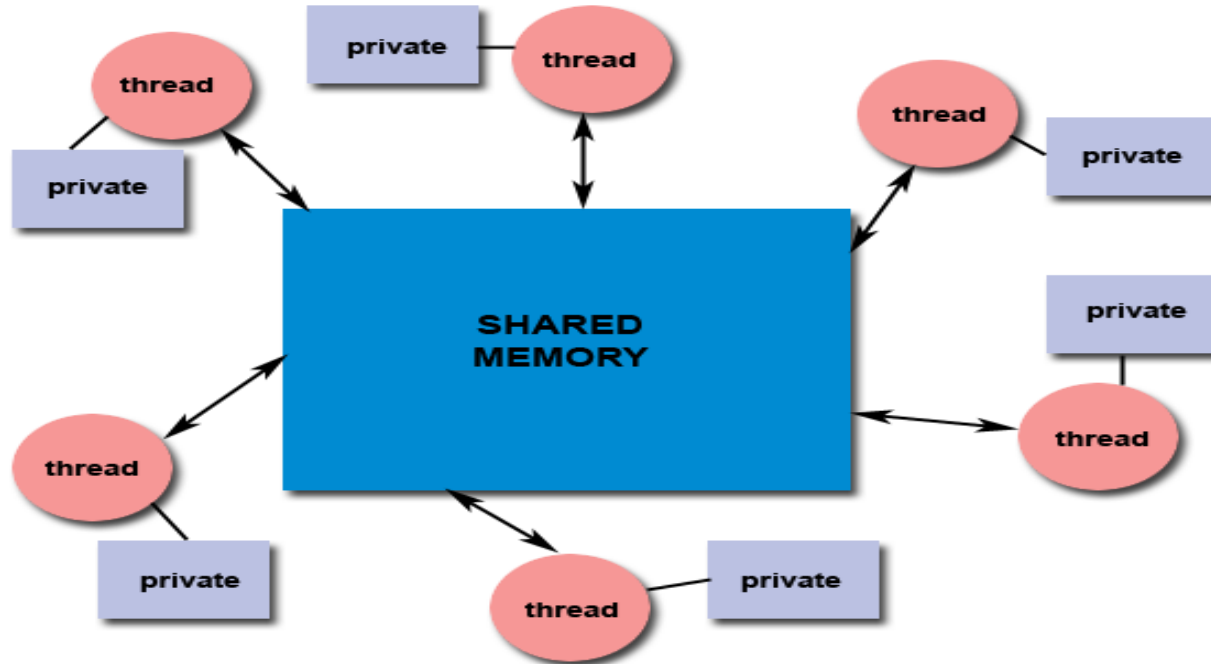
Thread creation and destruction less expensive than process creation and destruction

## Non-trivial

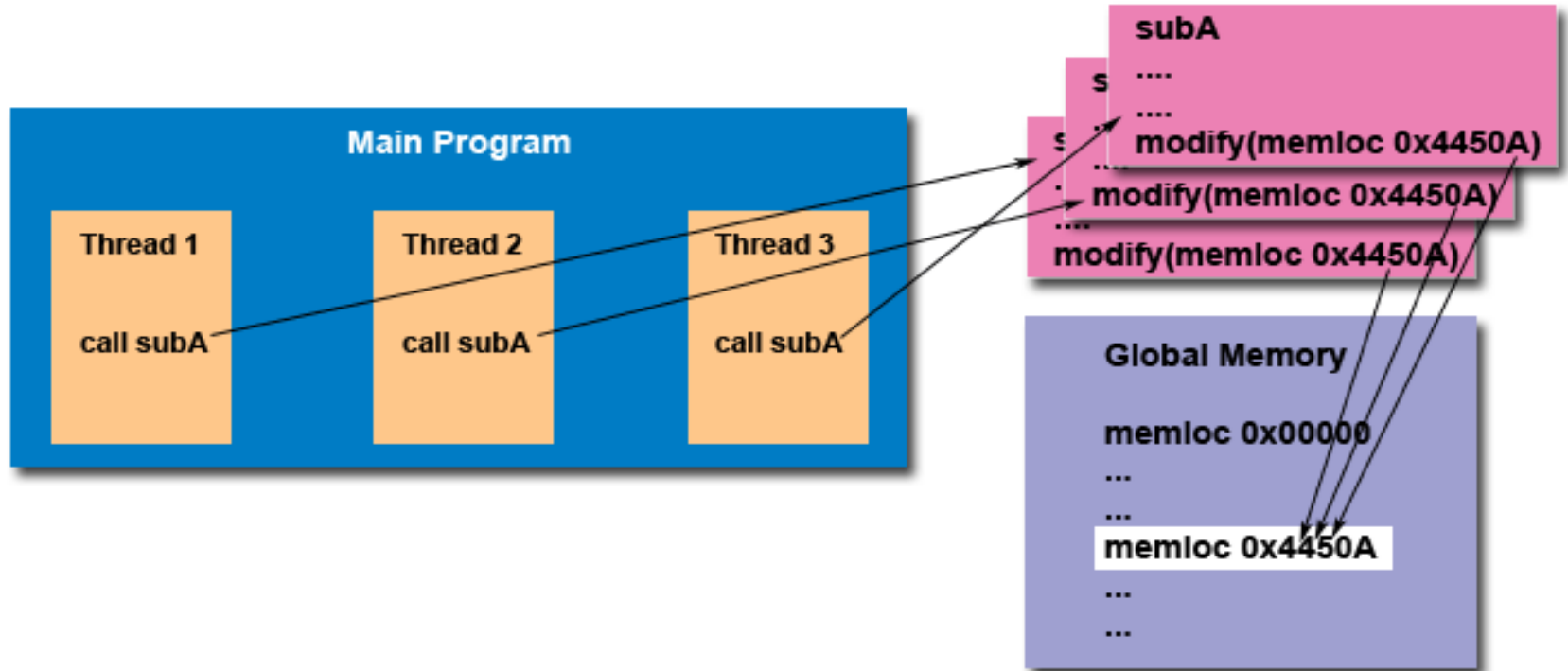
Have to prevent several threads from accessing and changing the same shared data at the same time (synchronization)



# Shared memory



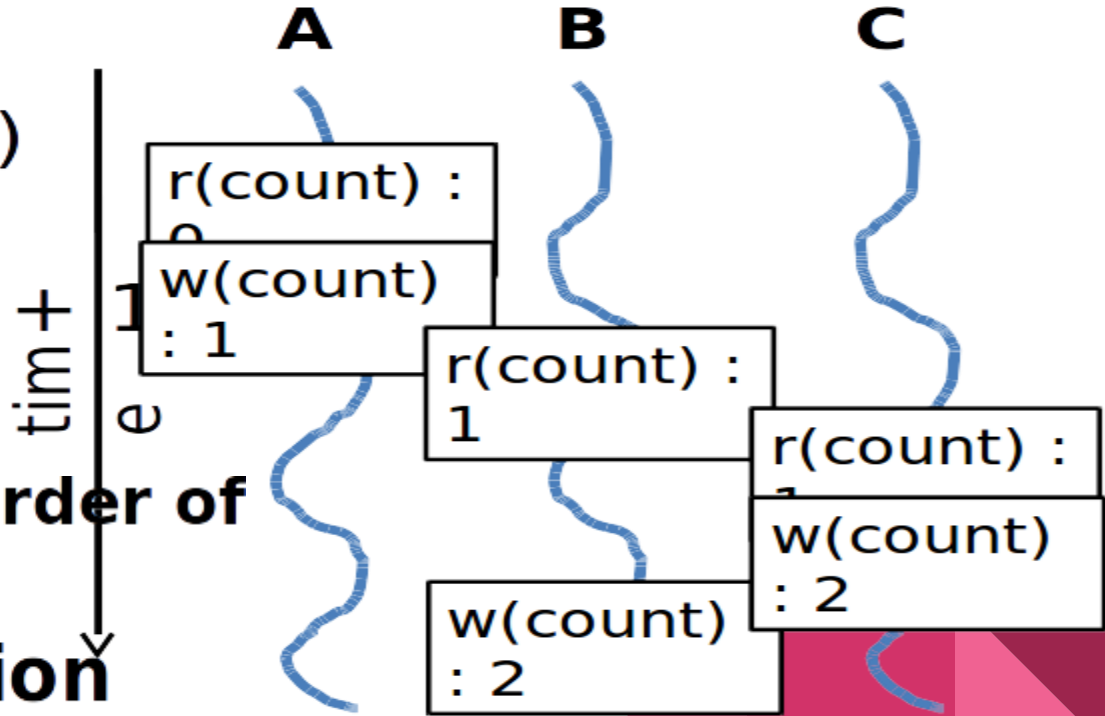
# Thread safety



# Race Condition

```
int count = 0;  
void increment()  
{  
    count = count + 1;  
}
```

**Result depends on order of execution**  
**=> Synchronization needed**




# Mutex

mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.

Mutexes are used for serializing shared resources.

Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it.

When a mutex lock is attempted against a mutex which is held by another thread, the thread is blocked until the mutex is unlocked.



# POSIX Threads

The POSIX thread libraries are a standards based thread API for C/C++.





# PThread Creation

```
int pthread_create(pthread_t * thread, const pthread_attr_t * attr,  
                  void * (*start_routine)(void *), void *arg);
```

## Arguments:

thread - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)

attr - Set to NULL if default thread attributes are used. void \* (\*start\_routine) - pointer to the function to be threaded. Function has a single argument: pointer to void.

\*arg - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

# Pthread exit


```
void pthread_exit(void *retval);
```

Arguments:

retval - Return value of thread.

This routine kills the thread.

Note: the return pointer \*retval, must not be of local scope otherwise it would cease to exist once the thread terminates.



# Pthread join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

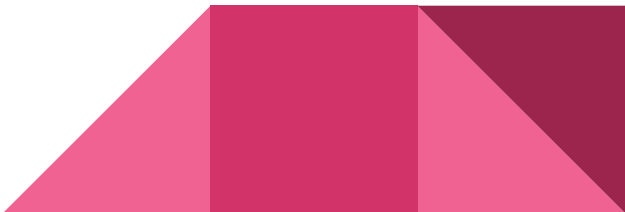
The *pthread\_join()* function suspends execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated.



# Mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The mutex object referenced by *mutex* is locked by calling *pthread\_mutex\_lock()*. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.



# Lab

Evaluate the performance of multithreaded sort

Add /usr/local/cs/bin to PATH

```
$ export PATH=/usr/local/cs/bin:$PATH
```

Generate a file containing 10M random double-precision floating point numbers, one per line with no white space

/dev/urandom: pseudo-random number generator



# Lab

## od

Write the contents of its input files to standard output in a user-specified format

### Options

- t f: Double-precision floating point

- N <count>: Format no more than count bytes of input

## sed, tr

Remove address, delete spaces, add newlines between each float

# Lab

Use `time -p` to time the command `sort -g` on the data you generated

Send output to `/dev/null`

Run `sort` with the `--parallel` option and the `-g` option: compare by general numeric value

Use `time` command to record the real, user and system time when running `sort` with 1, 2, 4, and 8 threads

```
$ time -p sort -g file_name > /dev/null (1 thread)
```

```
$ time -p sort -g --parallel=[2, 4, or 8] file_name > /dev/null
```

Record the times and steps in `log.txt`



# CS 35L

LAB 3,

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)



# Outline

**Security basics**


**Intrusion detection**

**Security policies**



# Communication Over the Internet

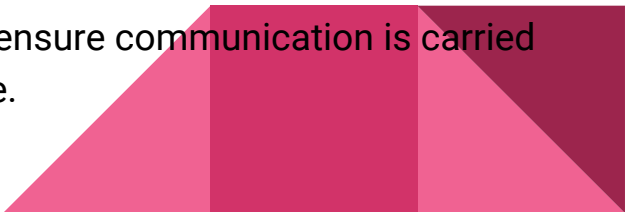
What type of guarantees do we want?

- Confidentiality
    - Message secrecy
  - Data integrity
    - Message consistency
  - Authentication
    - Identity confirmation
  - Authorization
    - Specifying access rights to resources
  - Availability
    - Information must be available to authorized individuals when they need it.
- 

# Telnet

A network protocol that allows a user on one computer to log onto another computer that is part of the same network.

Provides a bidirectional interactive text-oriented communication facility using a virtual terminal connection

- Telnet, by default, does not encrypt any data sent over the connection (including passwords), and so it is often feasible to eavesdrop on the communications and use the password later for malicious purposes; anybody who has access to a device located on the network between the two hosts where Telnet is being used can intercept the packets passing by and obtain login, password and whatever else is typed with a packet analyzer
  - Most implementations of Telnet have no authentication that would ensure communication is carried out between the two desired hosts and not intercepted in the middle.
- 

# What is SSH?

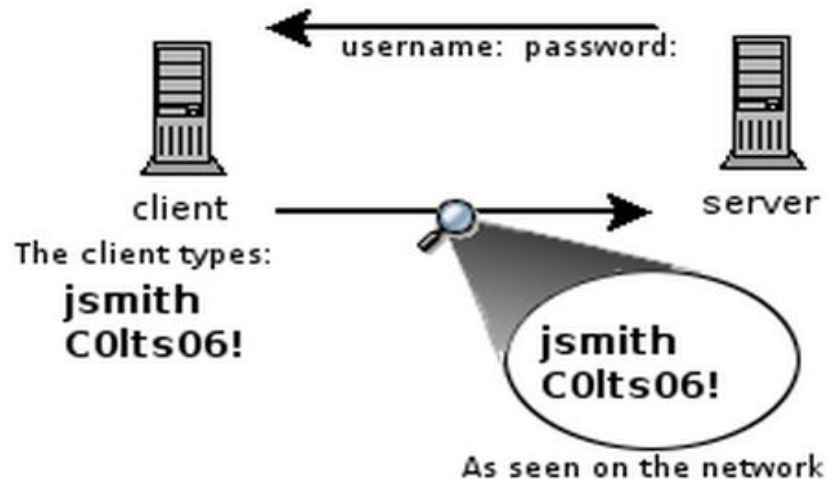
## Secure Shell

- Used to remotely access
- Successor of telnet
- Encrypted and better authenticated session

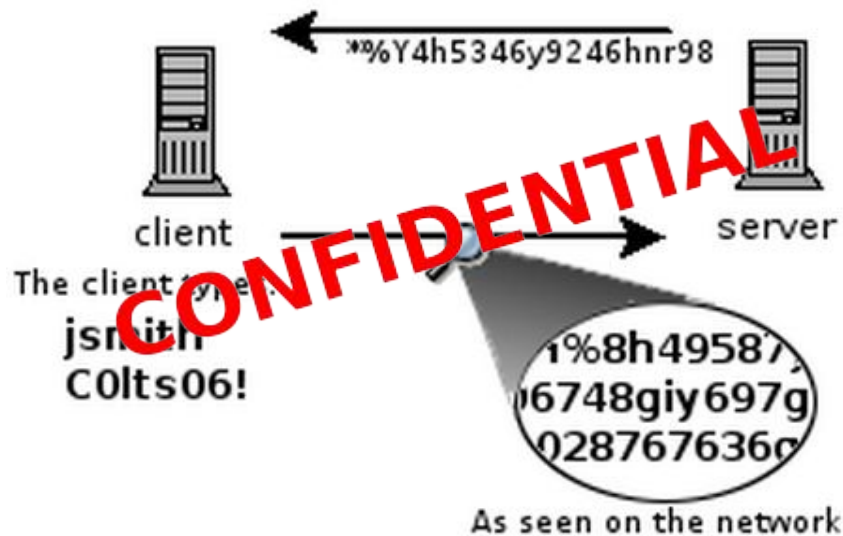


# What is SSH?

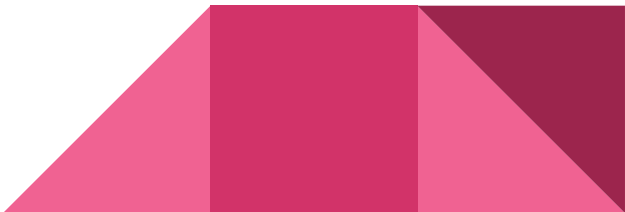
An unencrypted login session  
such as through telnet



An encrypted login session  
such as through SSH



# Encryption Types

- Symmetric Key Encryption
    - a.k.a shared/secret key
    - Key used to encrypt is the same as key used to decrypt
  - Asymmetric Key Encryption: Public/Private
    - 2 different (but related) keys: public and private
    - Only creator knows the relation. Private key cannot be derived from public key
  - Data encrypted with public key can only be decrypted by private key and vice versa
  - Public key can be seen by anyone
  - Never publish private key!!!
- 

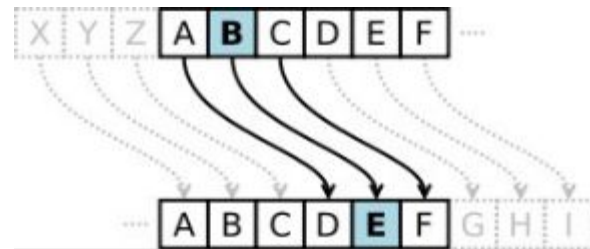
# Symmetric Key Encryption

Same secret key used for encryption and decryption

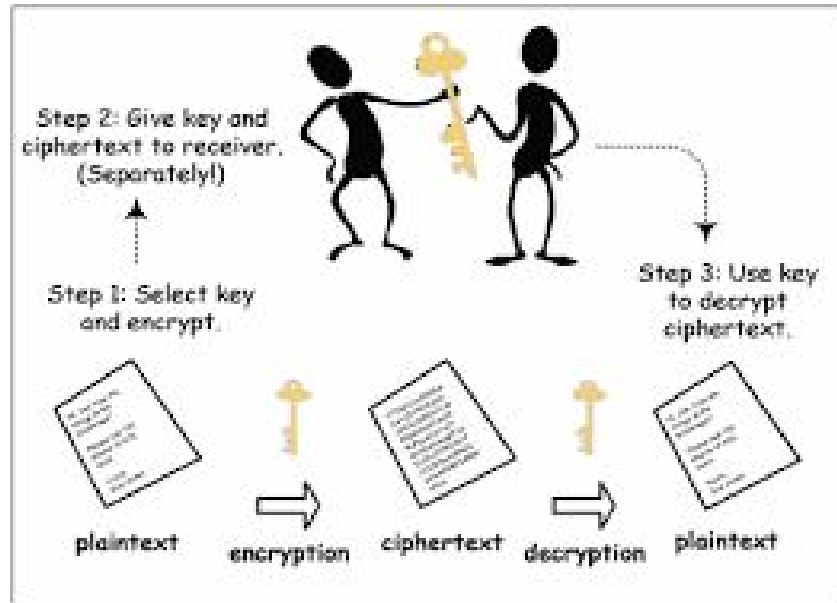
Example : Data Encryption Standard (DES)

Caesar's cipher

- Map the alphabet to a shifted version
  - Plaintext – SECRET.
  - Ciphertext – VHFUHW
  - Key is 3 (number of shifts of the alphabet)



# Symmetric Key Encryption



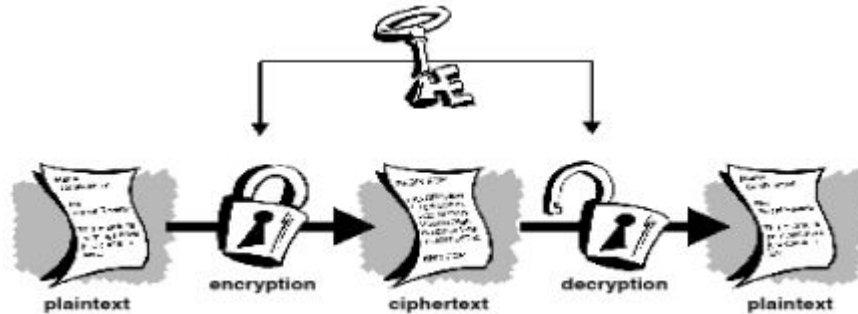


# Problem with Symmetric Key Encryption

Key distribution is a problem

The secret key has to be delivered in a safe way to the recipient

Chance of key being compromised



# Public-key Encryption (Asymmetric)

Uses a pair of keys for encryption

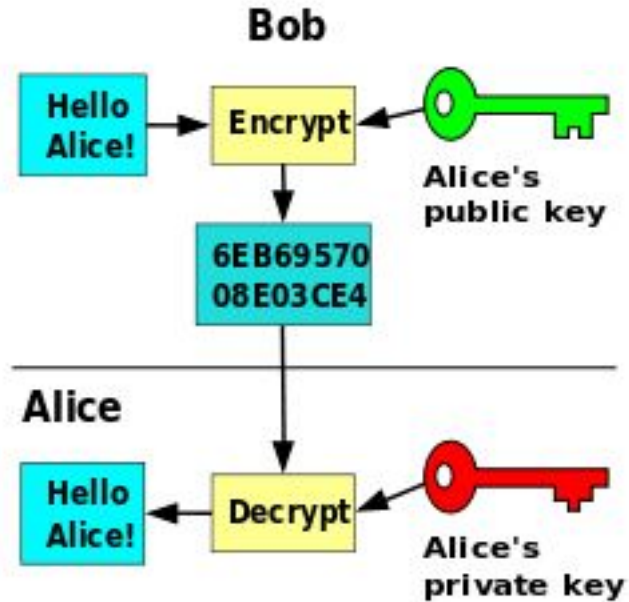
- Public key – Published and known to everyone
- Private key – Secret key known only to the owner
  - Encryption
    - Use public key to encrypt messages
    - Anyone can encrypt message, but they cannot decrypt the ciphertext
  - Decryption
    - Use private key to decrypt messages
  - Example: RSA



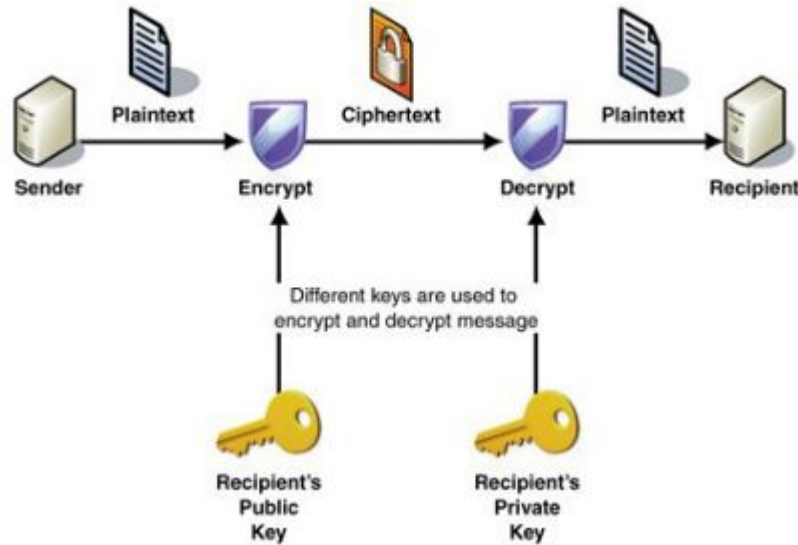
# Key generation



# Encryption



# Public-key Encryption (Asymmetric)



# SSH Protocol

Client ssh's to remote server

```
$ ssh username@somehost
```

If first time talking to server -> host validation

The authenticity of host 'somehost (192.168.1.1)' can't be established.

RSA key fingerprint is 90:9c:46:ab:03:1d:30:2c:5c:87:c5:c7:d9:13:5d:75.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'somehost' (RSA) to the list of known hosts.



# SSH Protocol

ssh doesn't know about this host yet

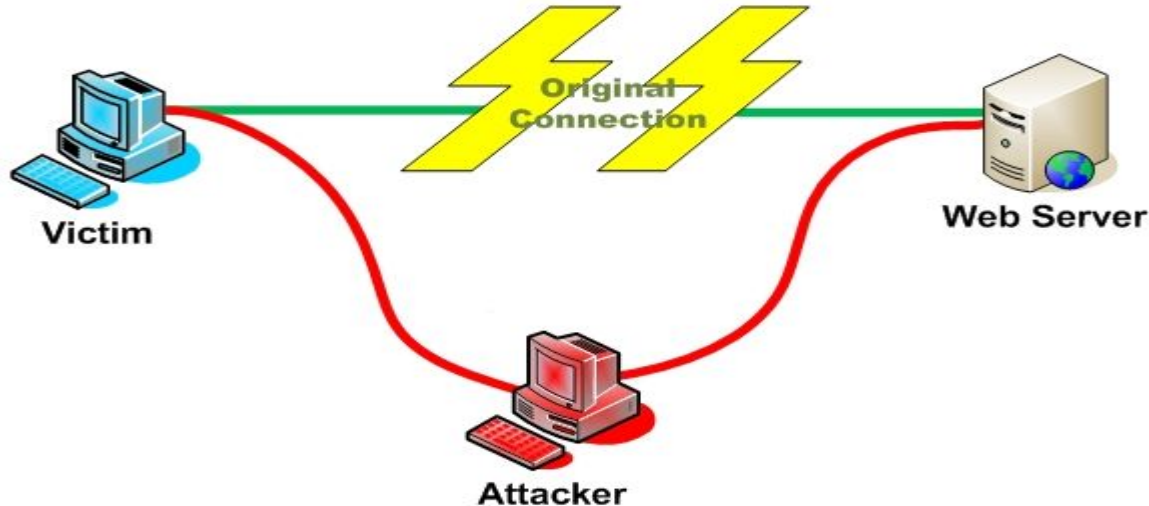
shows hostname, IP address and fingerprint of the server's public key, so you can be sure you're talking to the correct computer

After accepting, public key is saved in `~/.ssh/known_hosts`



# Host Validation

- Next time client connects to server
- Check host's public key against saved public key
- If they don't match, warning





# Host Validation

Client asks server to prove that it is the owner of the public key using asymmetric encryption

Encrypt a message with public key

If server is true owner, it can decrypt the message with private key

If everything works, host is successfully validated



# Session Encryption

Client and server agree on a symmetric encryption key (session key)

All messages sent between client and server

- encrypted at the sender with session key
- decrypted at the receiver with session key

Anybody who doesn't know the session key (hopefully, no one but client and server) doesn't know any of the contents of those messages



# Client Authentication

- Password-based authentication
  - Prompt for password on remote server
  - If username specified exists and remote password for it is correct then the system lets you in
- Key-based authentication
  - Generate a key pair on the client
  - Copy the public key to the server (`~/.ssh/authorized_keys`)
  - Server authenticates client if it can demonstrate that it has the private key
  - The private key can be protected with a passphrase
  - Every time you ssh to a host, you will be asked for the passphrase (inconvenient!)



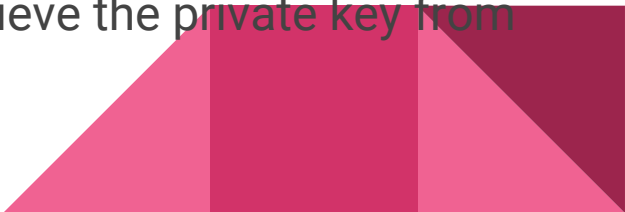
# ssh-agent (passphrase-less ssh)

A program used with OpenSSH that provides a secure way of storing the private key

ssh-add prompts user for the passphrase once and adds it to the list maintained by ssh-agent

Once passphrase is added to ssh-agent, the user will not be prompted for it again when using SSH

OpenSSH will talk to the local ssh-agent daemon and retrieve the private key from it automatically



# X Window System

Windowing system that forms the basis for most GUIs on UNIX

X is an architecture-independent system for remote graphical user interfaces and input device capabilities

X is a network-based system. It is based upon a network protocol such that a program can run on one computer but be displayed on another (X Session Forwarding)



# Lab

Securely log in to each others' computers

- Use ssh (OpenSSH)

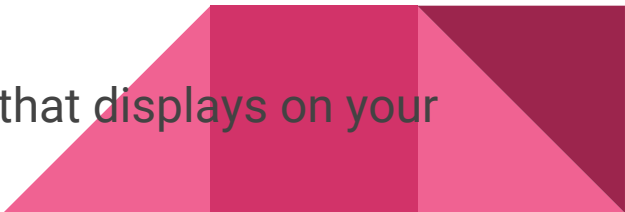
Use key-based authentication

- Generate key pairs

Make logins convenient

- Type your passphrase once and be able to use ssh to connect to any other host without typing any passwords or passphrases

Use port forwarding to run a command on a remote host that displays on your host



# Setup

## Ubuntu

Make sure you have openssh-server and openssh-client installed

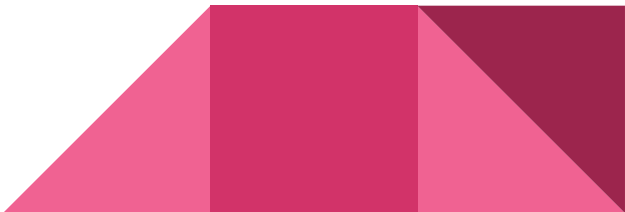
\$ dpkg --get-selections | grep openssh should output:

- openssh-server install
- openssh-client install

If not:

\$ sudo apt-get install openssh-server

\$ sudo apt-get install openssh-client



# Server Steps

Generate public and private keys

\$ ssh-keygen (by default saved to ~/.ssh/id\_rsa and id\_rsa.pub) – don't change the default location

Create an account for the client on the server

\$ sudo useradd -d /home/<homedir\_name> -m <username>

\$ sudo passwd <username>

Create .ssh directory for new user

\$ cd /home/<homedir\_name>





# Server steps

```
$ sudo mkdir .ssh
```

Change ownership and permission on .ssh directory

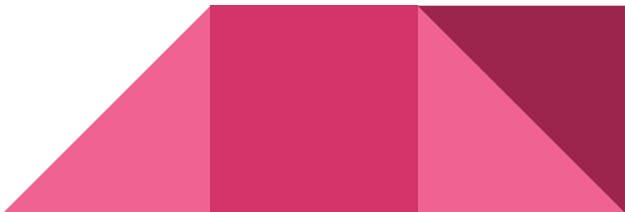
```
$ sudo chown -R username .ssh
```

```
$ sudo chmod 700 .ssh
```

Optional: disable password-based authentication

```
$ emacs /etc/ssh/sshd_config
```

change PasswordAuthentication option to no



# Client steps

Generate public and private keys

```
$ ssh-keygen
```

Copy your public key to the server for key-based authentication  
(`~/.ssh/authorized_keys`)

```
$ ssh-copy-id -i UserName@server_ip_addr
```

Add private key to authentication agent (`ssh-agent`)

```
$ ssh-add
```



# Client steps

SSH to server

```
$ ssh UserName@server_ip_addr
```

```
$ ssh -X UserName@server_ip_addr (X11 session forwarding)
```

Run a command on the remote host

```
$ xterm, $ gedit, $ firefox, etc.
```



# How to check IP Address

- `$ ifconfig`
  - configure or display the current network interface configuration information (IP address, etc.)
- `$ ping <ip_addr>(packet internet groper)`
  - Test the reachability of a host on an IP network
  - measure round-trip time for messages sent from a source to a destination computer
  - Example: `$ ping 192.168.0.1`, `$ ping google.com`



# Assignment 10 report format

1. Introduction
2. Background
3. Summary
4. Results
5. Your opinion



# Finals

- 100 points
- 180 minutes
- Open book + open notes
- Theory questions
- Programming questions (No coding on lab machines)
- No electronic gadgets allowed



# Digital Signature

An electronic stamp or seal - almost exactly like a written signature, except more guarantees!

Is appended to a document

- Or sent separately (detached signature)

Ensures data integrity

- Document was not changed during transmission



# Steps for generating a digital signature

SENDER:

## 1. Generate a Message Digest

- a. The message digest is generated using a set of hashing algorithms
- b. A message digest is a 'summary' of the message we are going to transmit
- c. Even the slightest change in the message produces a different digest

## 2. Create a Digital Signature

- a. The message digest is encrypted using the sender's private key. The resulting encrypted message digest is the digital signature

## 3. Attach digital signature to message and send to receiver





# Steps for generating a digital signature

RECEIVER:

1. Recover the Message Digest

- a. Decrypt the digital signature using the sender's public key to obtain the message digest generated by the sender

2. Generate the Message Digest

- a. Use the same message digest algorithm used by the sender to generate a message digest of the received message



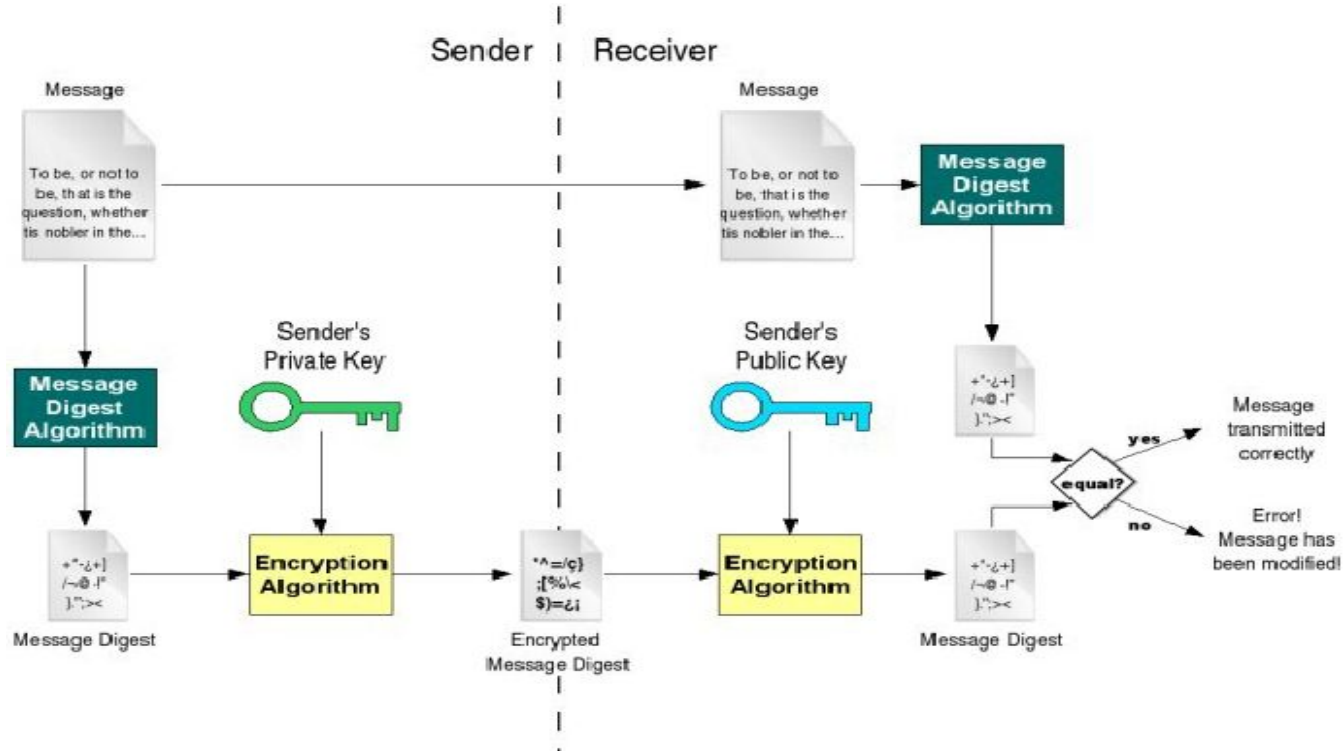
# Steps for generating a digital signature

RECEIVER:

1. Compare digests (the one sent by the sender as a digital signature, and the one generated by the receiver)
  - a. If they are not exactly the same => the message has been tampered with by a third party
  - b. We can be sure that the digital signature was sent by the sender (and not by a malicious user) because only the sender's public key can decrypt the digital signature and that public key is proven to be the sender's through the certificate. If decrypting using the public key renders a faulty message digest, this means that either the message or the message digest are not exactly what the sender sent.



# Digital Signature



# Detached Signature

Digital signatures can either be attached to the message or detached

A detached signature is stored and transmitted separately from the message it signs

Commonly used to validate software distributed in compressed tar files

You can't sign such a file internally without altering its contents, so the signature is created in a separate file



# Homework

Answer 2 questions in the file hw.txt

1. Generate a key pair with the GNU Privacy Guard's commands
  - a. `$ gpg --gen-key` (choose default options)
2. Export public key, in ASCII format, into hw-pubkey.asc
  - a. `$ gpg --armor --output hw-pubkey.asc --export 'Your Name'`



# Homework

- Make a tarball of the above files + log.txt and zip it with gzip to produce hw.tar.gz
  - `$ tar -cf hw.tar <files>`
  - `$ gzip hw.tar -> creates hw.tar.gz`
- Use the private key you created to make a detached clear signature hw.tar.gz.sig for hw.tar.gz
  - `$ gpg --armor --output hw.tar.gz.sig --detach-sign hw.tar.gz`
- Use given commands to verify signature and file formatting
- These can be found at the end of the assignment spec



# CS 35L

LAB 8,

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

**Linking and loading**

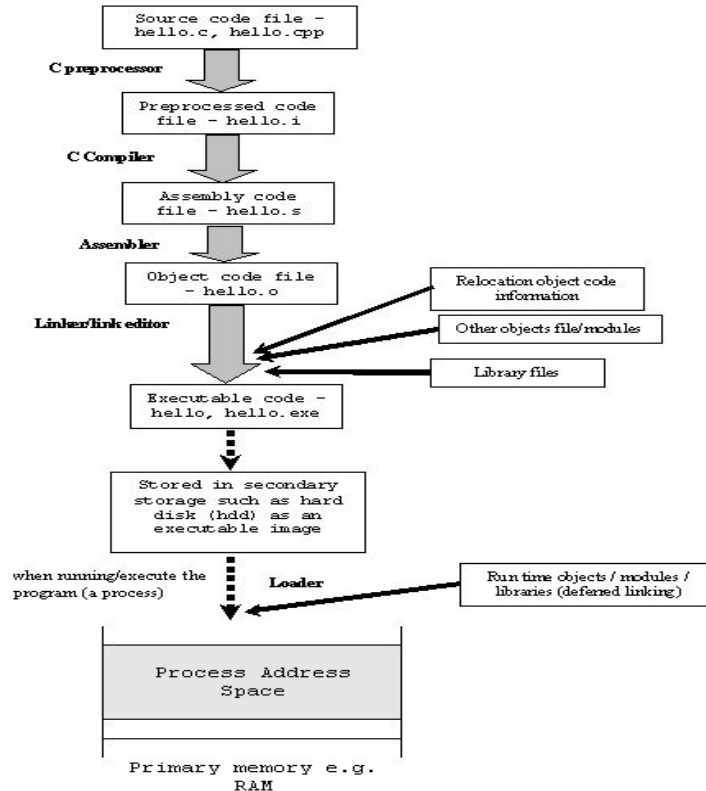
**Static linking**

**Dynamic linking**





# Building an executable file



# Linking and loading

Linker collects procedures and links object modules together into one executable program

Why isn't everything written as just one big program, saving the necessity of linking?

- Efficiency: if just one function is changed in a 100K line program, why recompile the whole program? Just recompile the one function and relink.
- Multiple-language programs
- Other reasons?



# Static linking

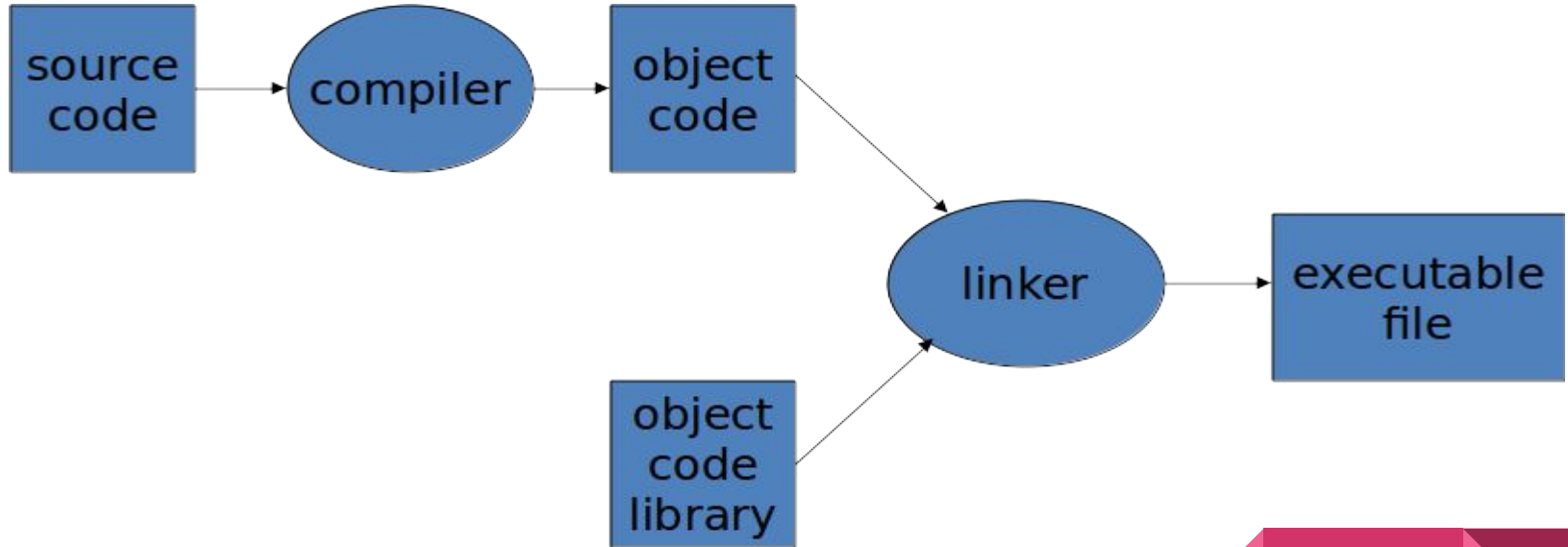
Carried out only once to produce an executable file

If static libraries are called, the linker will copy all the modules referenced by the program to the executable

Static libraries are typically denoted by the .a file extension



# Static linking



A previously compiled  
collection of standard  
program functions

# Dynamic linking

Allows a process to add, remove, replace or relocate object modules during its execution.

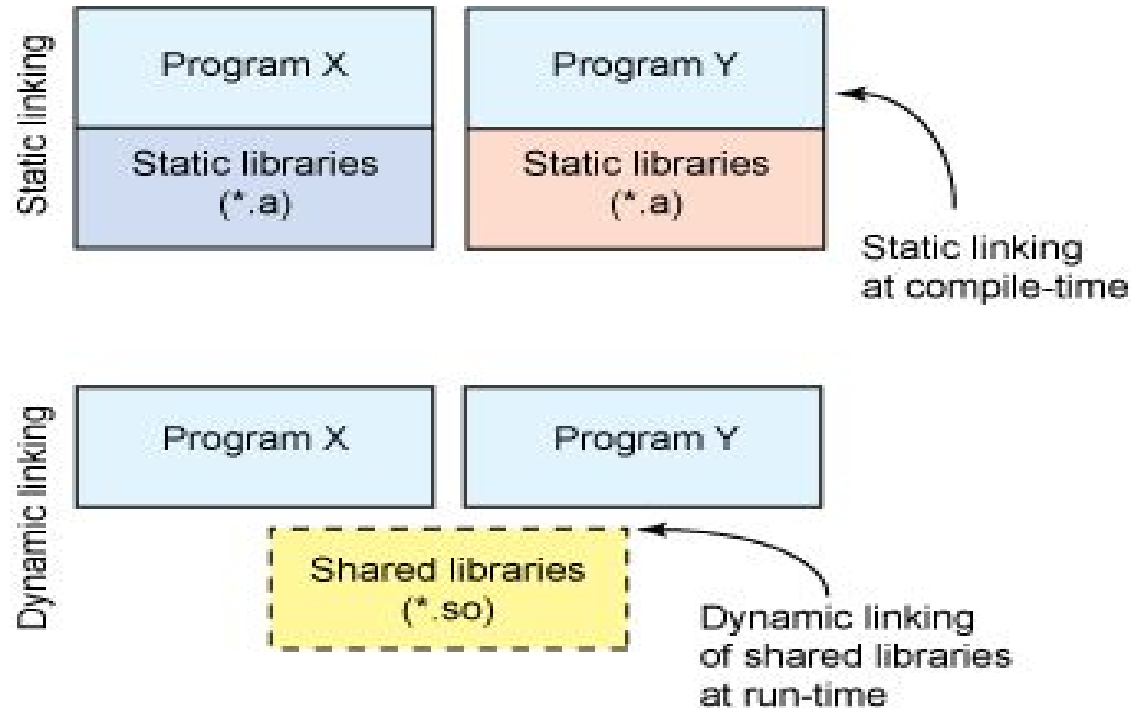
If shared libraries are called:

- Only copy a little reference information when the executable file is created
- Complete the linking during loading time or running time

Dynamic libraries are typically denoted by the .so file extension

- .dll on Windows
- 

# Dynamic linking



# Dynamic linking

Dynamic vs. static linking resulting size

If you are the sysadmin, which do you prefer?



# Advantages of dynamic linking

The executable is typically smaller

When the library is changed, the code that references it does not usually need to be recompiled.

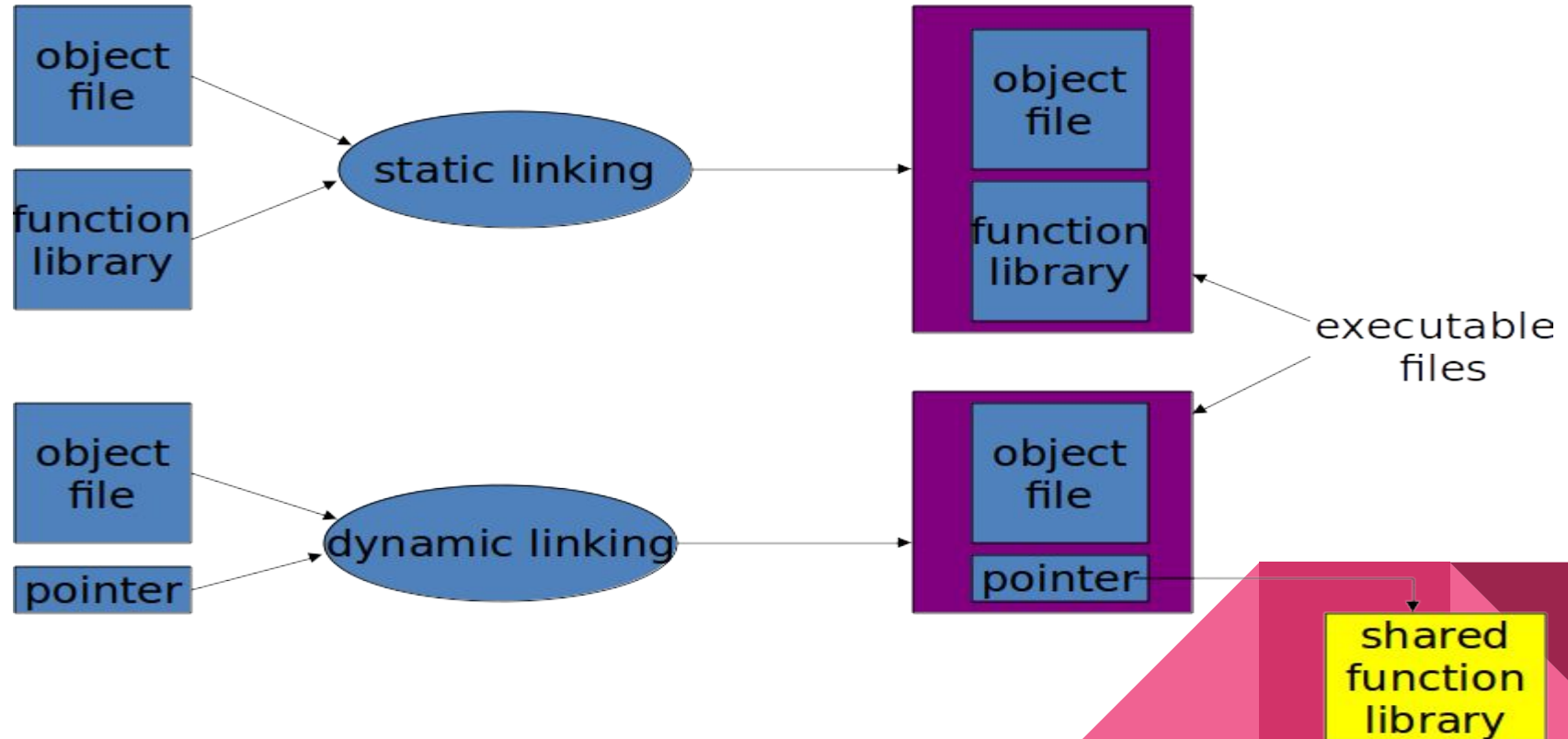
The executable accesses the .so at run time; therefore, multiple programs can access the same .so at the same time

- Memory footprint amortized across all programs using the same .so





# Smaller is more efficient



# Disadvantages of dynamic linking

## Performance hit

- Need to load shared objects (at least once)
- Need to resolve addresses (once or every time)

What if the necessary dynamic library is missing?

What if we have the library, but it is the wrong version?



# Lab 9

Write and build simple “`cos(sqrt(3.0))`” program in C

- Use `ldd` to investigate which dynamic libraries your `cos` program loads
- Use `strace` to investigate which system calls your `cos` program makes



# Lab 9

Use `"ls /usr/bin | awk 'NR%101==SID%101'"` to find ~25 linux commands to use ldd on

- Record output for each one in your log and investigate any errors you might see
- From all dynamic libraries you find, create a sorted list
  - Remember to remove the duplicates!



# CS 35L

LAB 8,

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

**Static linking**

**Dynamic linking**



# Gcc flags

-fPIC: Compiler directive to output position independent code, a characteristic required by shared libraries.

-lXXX: Link with "libXXX.so"

Without -L to directly specify the path, /usr/lib is used.

-L: At compile time, find .so from this path.



# Gcc flags

-Wl : passes options to linker

rpath : -rpath at runtime finds .so from this path.

-c: Generate object code from c code.

-shared: Produce a shared object which can then be linked with other objects to form an executable.

<https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html#Link-Options>





# How are libraries dynamically loaded?

**Table 1. The DI API**

Function	Description
<b>dlopen</b>	Makes an object file accessible to a program
<b>dlsym</b>	Obtains the address of a symbol within a dlopened object file
<b>dlerror</b>	Returns a string error of the last error that occurred
<b>dlclose</b>	Closes an object file

# Sample program

calc\_mean.h

```
double mean(double, double);
```



# Sample program

Calc\_mean.c

```
double mean(double a, double b) {  
    return (a+b)/2;  
}
```



# Sample program

```
#include <stdio.h>
```

```
#include "calc_mean.h"
```

```
int main(int argc, char* argv[]) {
```

```
    double v1, v2, m; v1 = 5.2; v2 = 7.9;
```

```
    m = mean(v1, v2);
```

```
    printf("The mean of %3.2f and %3.2f is %3.2f\n", v1, v2, m);
```

```
    return 0; }
```



# Creating and using a static library

```
gcc -c calc_mean.c -o calc_mean.o
```

```
ar rcs libmean.a calc_mean.o
```

```
gcc -static main.c -L. -lmean -o statically_linked
```



# Creating and using a dynamic library

```
gcc -c -fPIC calc_mean.c -o calc_mean.o
```

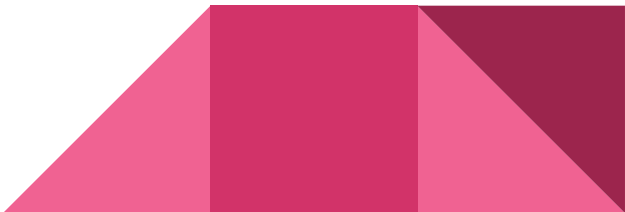
```
gcc -shared -o libmean.so calc_mean.o
```

```
gcc calc_mean.c -o dynamically_linked -L. -lmean
```

```
LD_LIBRARY_PATH=/u/cs/class/cs35l/cs35lt9/Documents/lab8
```

`./dynamically_linked` (**LD\_LIBRARY\_PATH** is an environment variable you set to give the run-time shared library loader (ld.so) an extra set of directories to look for when searching for shared libraries. Multiple directories can be listed, separated with a colon (:))

```
gcc main.c -o dynamically_linked -L. -lmean  
-Wl,-rpath=/u/cs/class/cs35l/cs35lt9/Documents/lab8
```



# Creating and using a dynamic library

```
gcc main.c -o dynamically_linked -L. -lmean
```

```
-Wl,-rpath=/u/cs/class/cs35l/cs35lt9/Documents/lab8
```

**rpath** - Add a directory to the runtime library search path. This is used when linking an ELF executable with shared objects. All **-rpath** arguments are concatenated and passed to the runtime linker, which uses them to locate shared objects at runtime.



# Why -L and -rpath ?

The linker needs to know what symbols should be supplied by the dynamic library.

For this reason, -L is needed to specify where the file to link against is.

-rpath comes into play at runtime, when the application tries to load the dynamic library. It informs the program of an additional location to search in when trying to load a dynamic library.






# -fPIC and -shared

-shared (linker option) - Produce a shared object which can then be linked with other objects to form an executable. For predictable results, you must also specify the same set of options used for compilation (-fpic, -fPIC) when you specify this linker option

-fpic (compile time option) - Generate position-independent code (PIC) suitable for use in a shared library, if supported for the target machine. Such code accesses all constant addresses through a global offset table (GOT). The dynamic loader resolves the GOT entries when the program starts (the dynamic loader is not part of GCC; it is part of the operating system)



# Attributes of a function

Used to declare certain things about functions called in your program

Help the compiler optimize calls and check code

Also used to control memory placement, code generation options or call/return conventions within the function being annotated

Introduced by the attribute keyword on a declaration, followed by an attribute specification inside double parentheses



# Attributes of a function

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully.

The keyword `__attribute__` allows you to specify special attributes when making a declaration

- `__attribute__((__constructor__))`
  - Is run when `dlopen()` is called
- `__attribute__((__destructor__))`
  - Is run when `dlclose()` is called



# Attributes of a function

Example:

```
__attribute__((__constructor__))
```

```
void to_run_before (void) {
```

```
    printf("pre_func\n");
```

```
}
```



# Homework

Split randall.c into 4 separate files

Stitch the files together via static and dynamic linking to create the program

randmain.c must use dynamic loading, dynamic linking to link up with randlibhw.c and randlibsw.c (using randlib.h)

Write the randmain.mk makefile to do the linking



# Homework

- randall.c outputs N random bytes of data
- Look at the code and understand it; Helper functions that check if hardware random number generator is available, and if it is, generates number
  - Hw RNG exists if RDRAND instruction exists
  - Uses cpuid to check whether CPU supports RDRAND (30th bit of ECX register is set)
- Helper functions to generate random numbers using software implementation (/dev/urandom)
- main function
  - Checks number of arguments (name of program, N)
  - Converts N to long integer, prints error message otherwise
  - Uses helper functions to generate random number using hw/sw



# Homework

- Divide randall.c into dynamically linked modules and a main program; Don't want resulting executable to load code that it doesn't need (dynamic loading)
  - randcpuid.c: contains code that determines whether the current CPU has the RDRAND instruction. Should include randcpuid.h and implement interface described by it.
  - randlibhw.c: contains the hardware implementation of the random number generator. Should include randlib.h and implement the interface described by it.
  - randlibsw.c: contains the software implementation of the random number generator. Should include randlib.h and implement the interface described by it.
  - randmain.c: contains the main program that glues together everything else. Should include randcpuid.h (as the corresponding module should be linked statically) but not randlib.h (as the corresponding module should be linked after main starts up). Depending on whether the hardware supports the RDRAND instruction, this main program should dynamically load the hardware-oriented or software-oriented implementation of randlib.

# CS 35L

LAB 8,

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)



# Outline

- Software version management
- Git
- Lab Assignment

## References:

<https://git-scm.com/>



# Software development process

- Involves making a lot of changes to code
  - New features added
  - Bugs fixed
  - Performance enhancements
- Software team has many people working on the same/different parts of code
- Many versions of software released
  - Ubuntu 10, Ubuntu 12, etc
  - Need to be able to fix bugs for Ubuntu 10 for customers using it, even though you have shipped Ubuntu 12.

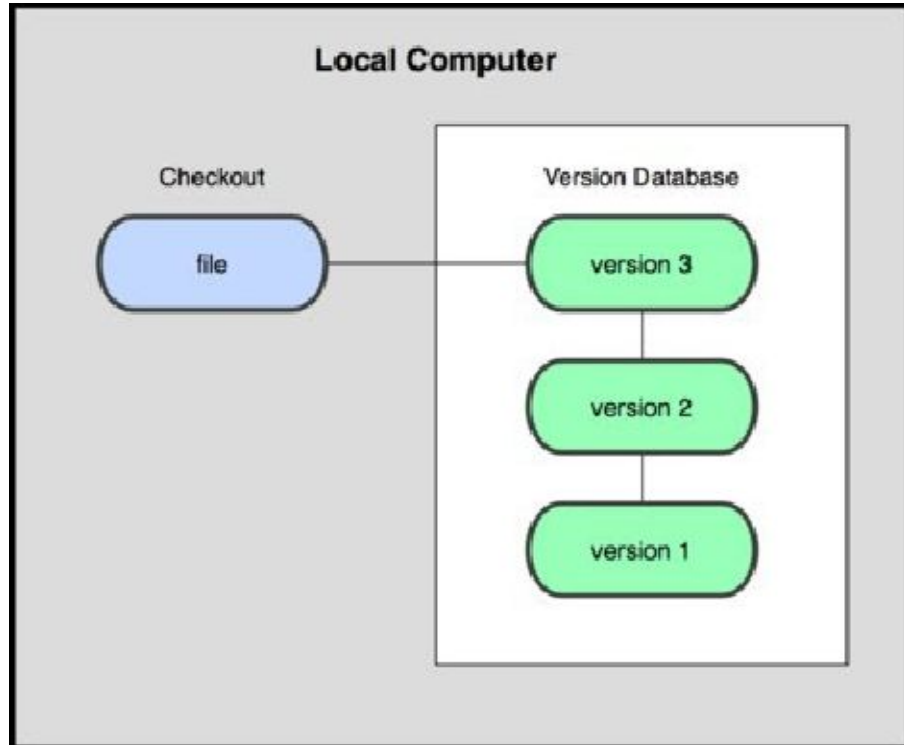


# Version control

- Track changes to code and other files related to the software
  - What new files were added?
  - What changes made to files?
  - Which version had what changes?
  - Which user made the changes?
- Track entire history of the software
- Version control software
  - Examples: GIT, Subversion, Perforce



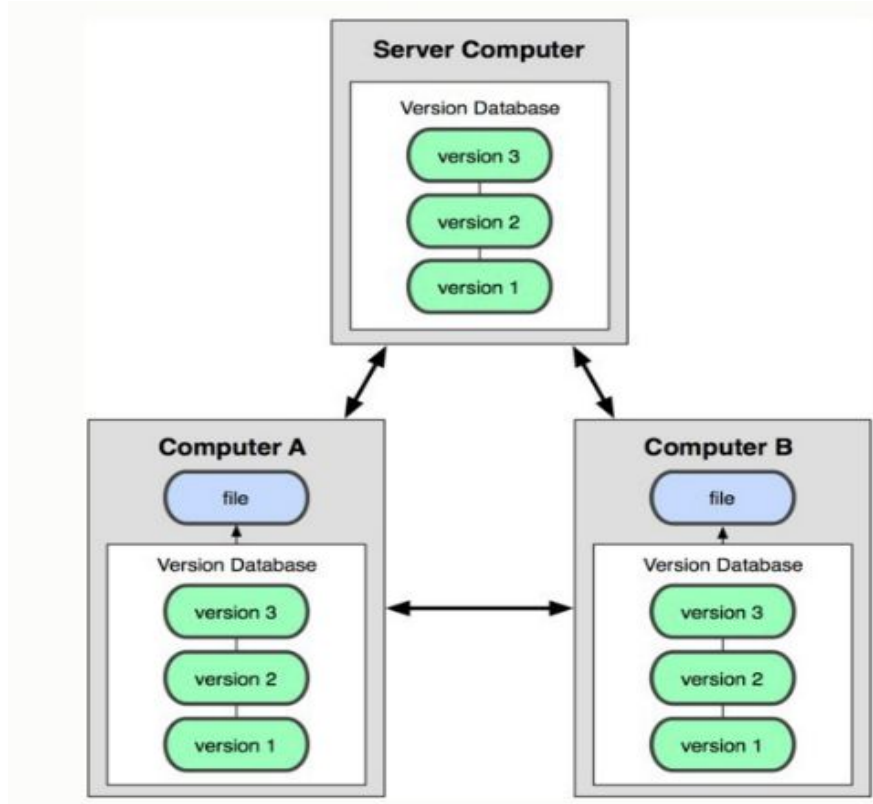
# Local VCS



- Organize different versions as folders on the local machine
- No server involved
- Other users should copy it via disk/network

Image Source: [git-scm.com](https://git-scm.com)

# Distributed VCS



- Version history is replicated at every user's machine
- Users have version control all the time
- Changes can be communicated between users
- Git is distributed

# Technical Terms

- Repository
  - Files and folder related to the software code
  - Full History of the software
- Working copy
  - Copy of software's files in the repository
- Check out
  - To create a working copy of the repository
- Check in/ Commit
  - Write the changes made in the working copy to the repository
  - Commits are recorded by the VCS



# Source Control with GIT

# Git States

## Local Operations

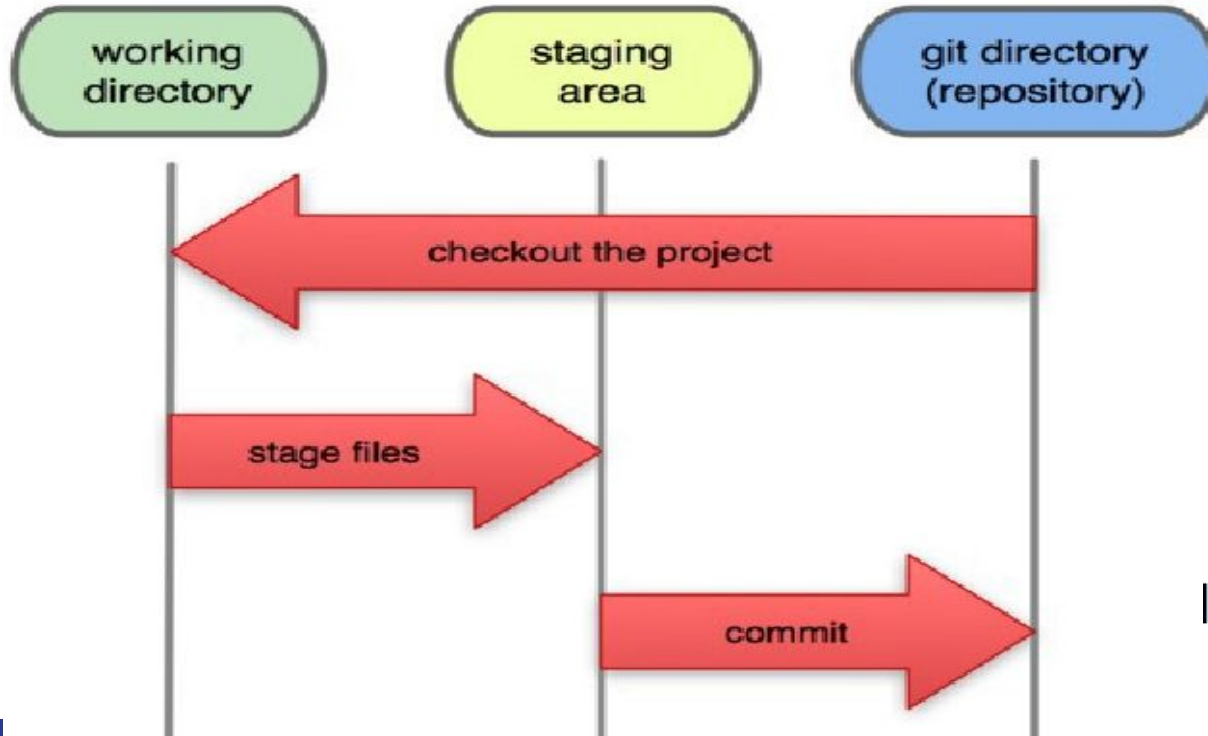


Image Source: [git-scm.com](https://git-scm.com)



# First Git Repository

- `mkdir gitroot`
- `cd gitroot`
- `git init`
  - creates an empty git repo (.git directory with all necessary subdirectories)
- `echo "Hello World" > hello.txt`
- `git status`
- `git add .`
  - Adds content to the index
  - Must be run prior to a commit
- `git commit -m 'Check in number one'`



# Git commands

- `echo "I love Git" >> hello.txt`
- `git status`
  - Shows list of modified files
- `git diff`
  - Shows changes we made compared to index
- `git add hello.txt`
- `git diff HEAD`
  - Now we can see changes in working version
- `git commit -m "Second commit"`



# Git commands

- Create Repo
  - `git init` (Create a new repo)
  - `git clone` (Create copy of existing repo)
- Branching
  - `git checkout <tag/commit> -b <new_branch_name>` (creates a new branch)
- Commits
  - `git add` (Stage modified/new files)
  - `git commit` (check-in the changes to the repository)



# Git commands

- Getting info
  - git status (Shows modified files, new files, etc)
  - git diff (compares working copy with staged files)
  - git log (Shows history of commits)
  - git show (Show a certain object in the repository)
- Getting help
  - git help



# Git Merge

A---B---C topic  
/  
D---E---F---G master

A---B---C topic  
/      \  
D---E---F---G---H master

Join two or more development histories

Merging hotfix branch into master

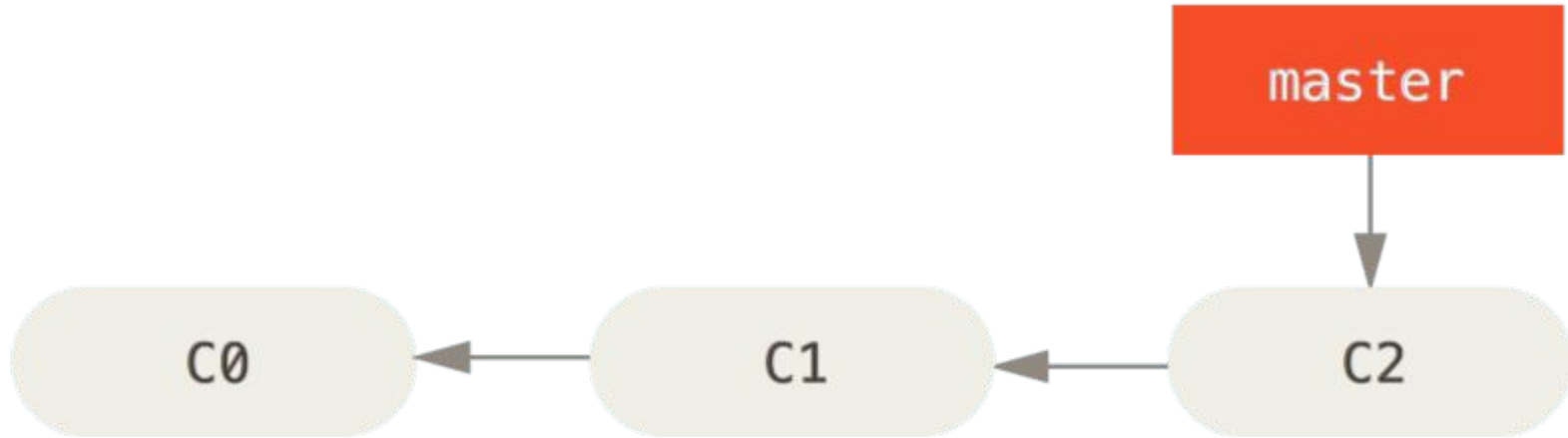
- git checkout master
- git merge topic
- Git tries to merge automatically

Simple if its a forward merge

Otherwise, you have to manually resolve conflicts

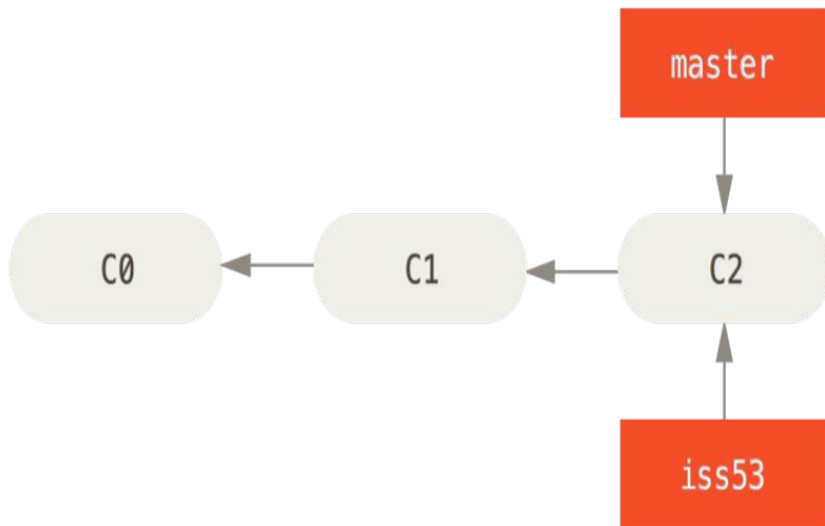
Image Source: [git-scm.com](https://git-scm.com)

# Branching and merging



A simple commit history

# Branching and merging

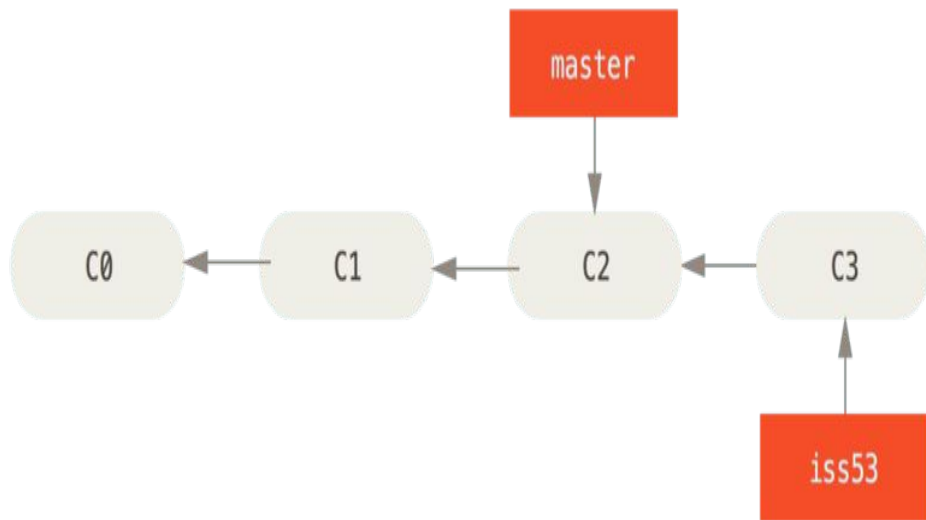


`$ git checkout -b iss53`  
Switched to a new branch "iss53"

This is shorthand for:

`$ git branch iss53`  
`$ git checkout iss53`

# Branching and merging

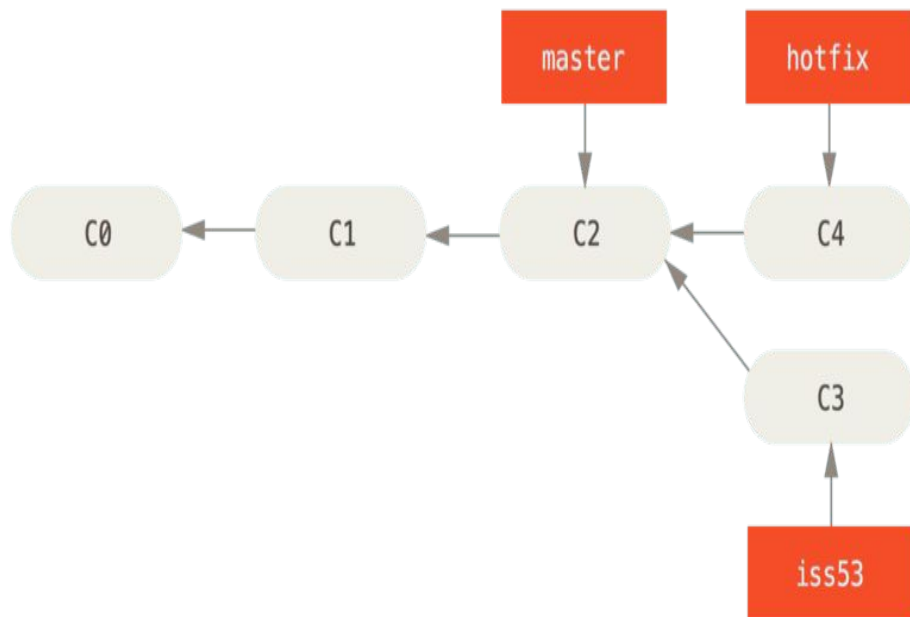


`vim index.html`

`git commit -a -m 'added a new footer [issue 53]'`



# Branching and merging



```
$ git checkout -b hotfix
```

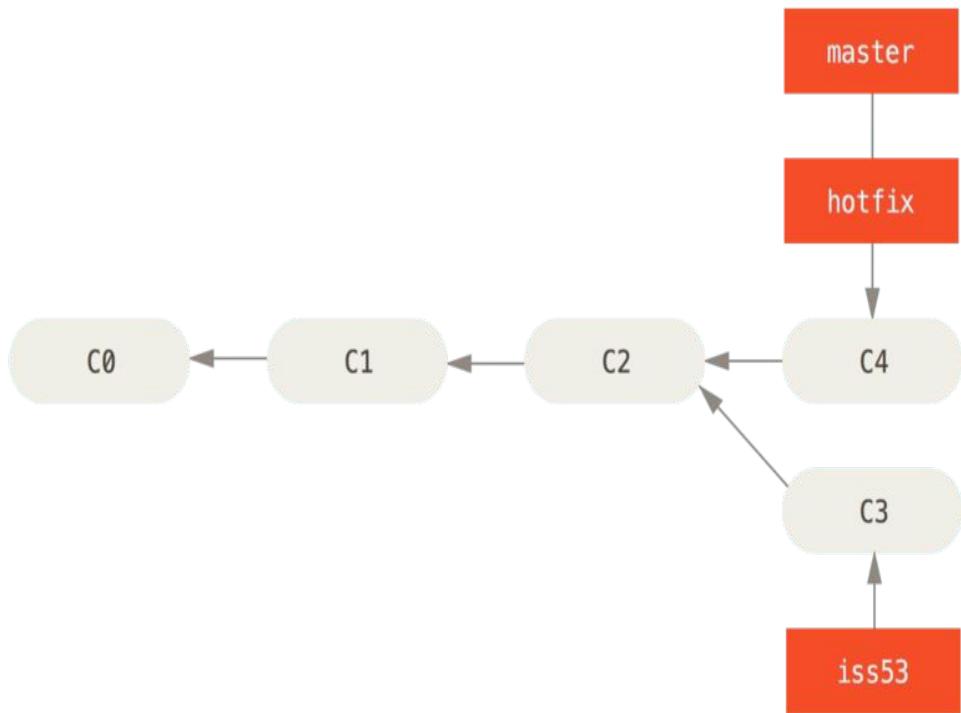
Switched to a new branch 'hotfix'

```
$ vim index.html
```

```
$ git commit -a -m 'fixed the broken email  
address'
```

```
[hotfix 1fb7853] fixed the broken email address  
1 file changed, 2 insertions(+)
```

# Branching and merging



```
$git checkout master
```

```
$ git merge hotfix
```

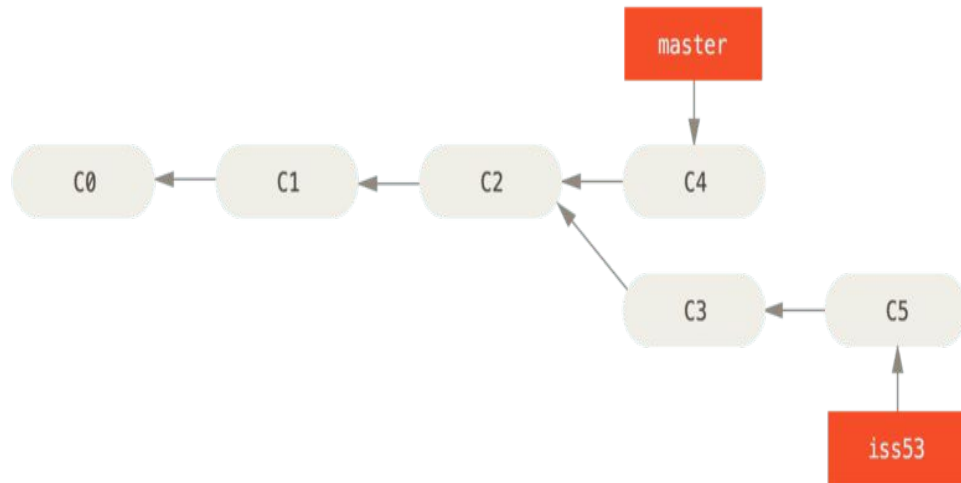
```
Updating f42c576..3a0874c
```

```
Fast-forward
```

```
index.html | 2 ++
```

```
1 file changed, 2 insertions(+)
```

# Branching and merging



```
git branch -d hotfix
```

```
$ git checkout iss53
```

Switched to branch "iss53"

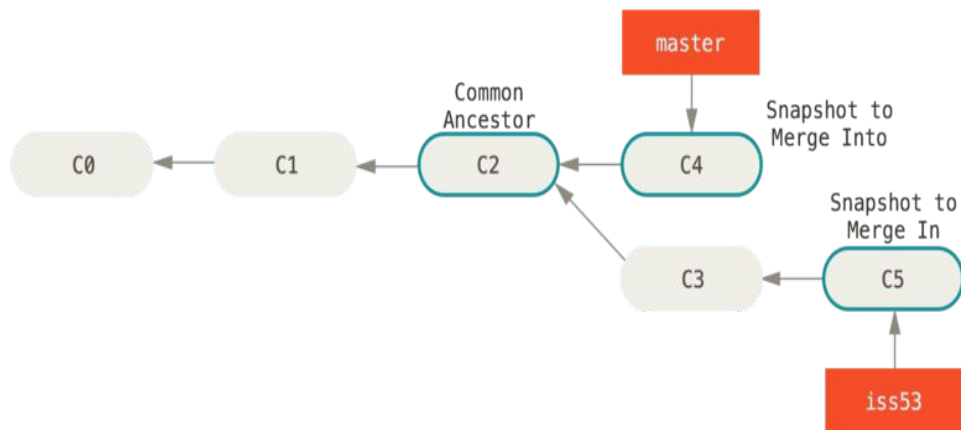
```
$ vim index.html
```

```
$ git commit -a -m 'finished the new footer  
[issue 53]'
```

```
[iss53 ad82d7a] finished the new footer [issue  
53]
```

```
1 file changed, 1 insertion(+)
```

# Branching and merging



git checkout master

Switched to branch 'master'

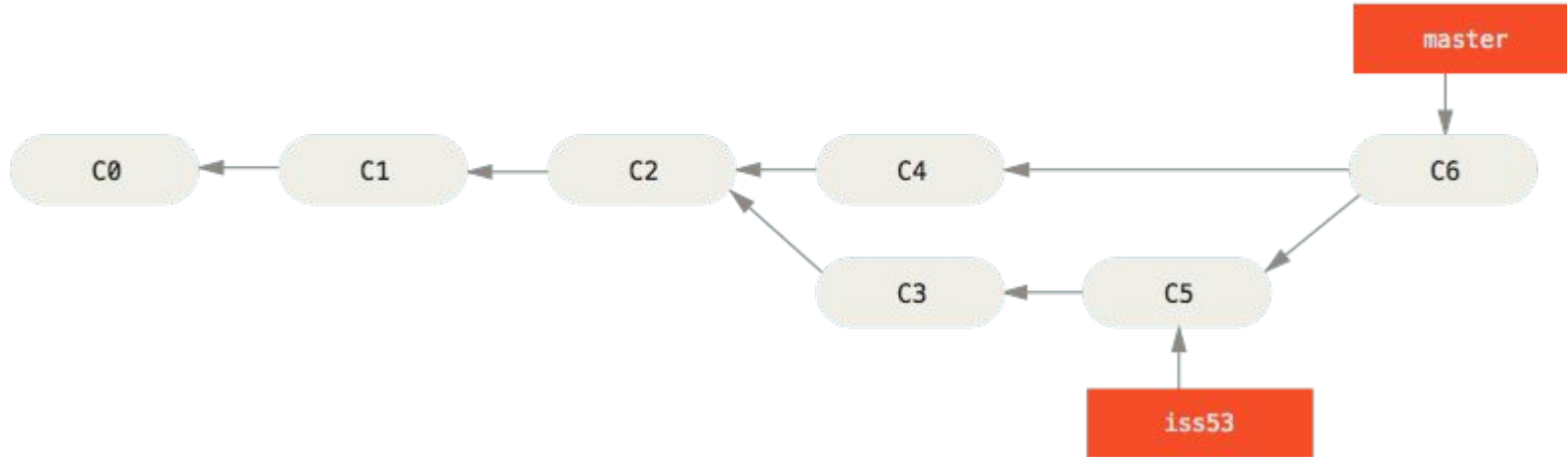
\$ git merge iss53

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)

# Branching and Merging



# Branching and Merging

```
$ git merge iss53
```

```
Auto-merging index.html
```

```
CONFLICT (content): Merge conflict in index.html
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

- Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run `git status`:

```
$ git status
```

```
On branch master
```


```
You have unmerged paths.
```

```
(fix conflicts and run "git commit")
```

```
Unmerged paths:
```

```
(use "git add <file>..." to mark resolution)
```

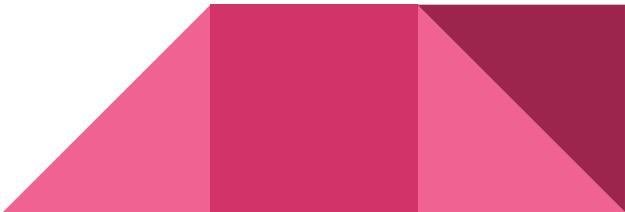
```
both modified:   index.html
```



# Branching and Merging

Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

```
<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> iss53:index.html
```

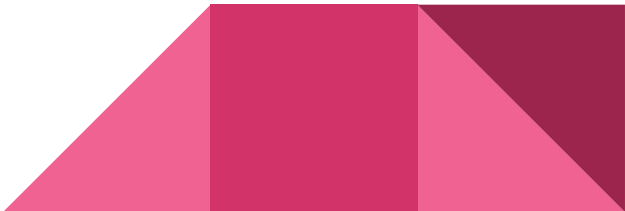


# Branching and Merging

This means the version in HEAD (your master branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the =====), while the version in your iss53 branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

This resolution has a little of each section, and the <<<<<<, =====, and >>>>>> lines have been completely removed. After you've resolved each of these sections in each conflicted file, run `git add` on each file to mark it as resolved. Staging the file marks it as resolved in Git.





# Branching and Merging

After you exit the merge tool, Git asks you if the merge was successful. If you tell the script that it was, it stages the file to mark it as resolved for you. You can run `git status` again to verify that all conflicts have been resolved:

```
$ git status
```

```
On branch master
```


```
All conflicts fixed but you are still merging.
```

```
(use "git commit" to conclude merge)
```

```
Changes to be committed:
```

```
    modified:   index.html
```

If you're happy with that, and you verify that everything that had conflicts has been staged, you can type `git commit` to finalize the merge commit.



# Branching and Merging

If you're happy with that, and you verify that everything that had conflicts has been staged, you can type `git commit` to finalize the merge commit. The commit message by default looks something like this:

```
Merge branch 'iss53'
```

```
Conflicts:
```

```
    index.html
```

```
# It looks like you may be committing a merge.
```

```
# If this is not correct, please remove the file
```

```
#      .git/MERGE_HEAD
```

```
# and try again.
```

```
# Please enter the commit message for your changes. Lines starting
```

```
# with '#' will be ignored, and an empty message aborts the commit.
```

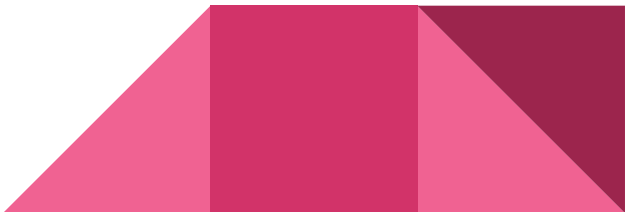
```
# On branch master
```

```
# All conflicts fixed but you are still merging.
```

```
#
```

```
# Changes to be committed:
```

```
#      modified:   index.html
```



# More Commands

- Reverting
  - `git checkout HEAD main.cpp`
    - Gets the HEAD revision for the working copy
  - `git checkout -- main.cpp`
    - Reverts changes in the working directory
  - `git revert`
    - Reverting commits (this creates new commits)
- Cleaning up untracked files
  - `git clean`
- Tagging
  - Human readable pointers to specific commits
  - `git tag -a v1.0 -m 'Version 1.0'`
    - This will name the HEAD commit as v1.0



# Lab Assignment

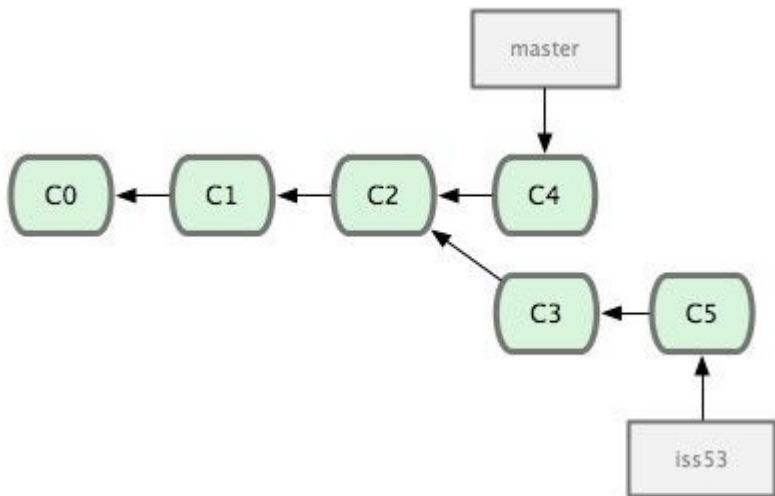
# Setup

- Installing Git
  - Ubuntu: `sudo apt-get install git`
  - SEASnet: Git is installed in `/usr/local/cs/bin`
- Add it to PATH variable or use whole path
  - `export PATH=/usr/local/cs/bin:$PATH`
- Make a directory 'gitroot' and get a copy of the Diffutils Git repository
  - `mkdir gitroot`
  - `cd gitroot`
  - `git clone git://git.savannah.gnu.org/diffutils.git`
- Use `man git` to find commands



# Hints

- Backporting
  - Apply a patch to a previous version
- Fix an issue with the diff diagnostic
- Hints
  - `git clone`
  - `git log`
  - `git tag`
  - `git show <hash_value>`
  - `git checkout v3.0 -b <BranchName>`



# CS 35L

LAB 8,

TA: Sucharitha Prabhakar

EMAIL ID: [prabhakarsucharitha@gmail.com](mailto:prabhakarsucharitha@gmail.com)

# Outline

- Diff
- Patch





# Diff

- Compares file line by line



# Diff - Example

- \$ cat file1

Hi,  
Hello,  
How are you?  
I am fine,  
Thank you.

\$ cat file2  
Hello,  
Hi,  
How are you?  
I am fine.

**diff file1 file2**

1d0

< Hi,

2a2

> Hi,

4,5c4

< I am fine,

< Thank you.

---

> I am fine.

# Diff - Example

- The first argument to diff command is regarded as old file while the second argument becomes new file.
- Expressions like 1d0 2a2, 4,5c4 can be decoded with the syntax **[line number or range from old file][action][line number or range from new file]**. Where, 'action' can be append, delete or changed-so-replace.
- The mark < represents the line to be deleted while > represents the line to be added.



# Diff - Example

- Use -i to ignore case differences
- Report That The Files Are Same Using -s Option
- Use -b To Ignore Spaces
- diff command can also be used to compare two directories



# Patch

- Download the patch which is typically a fix to a software
- Apply the patch
- Compile
- Install



# Patch

- Create a patch file using a diff
- Apply patch using patch command



# Patch

- Create a patch from a Source Tree
- Apply patch file to a source code tree



# Patch

- Take a backup before applying the patch using -b
  - `$ patch -b < hello.patch`
  - To decide backup file format: `patch -b -V numbered < hello.patch`
- Dry-run patch (To check if you will get any errors)
  - `patch --dry-run < hello.patch`
- Undo a patch
  - `patch -R < hello.patch`





# Makefile

- Simple way to organize compilation



# Makefile - Example

hellomake.c	hellofunc.c	hellomake.h
<pre>#include &lt;hellomake.h&gt;  int main() {     // call a function in another file     myPrintHelloMake();      return(0); }</pre>	<pre>#include &lt;stdio.h&gt; #include &lt;hellomake.h&gt;  void myPrintHelloMake(void) {     printf("Hello makefiles!\n");      return; }</pre>	<pre>/* example include file */ void myPrintHelloMake(void);</pre>

gcc -o hellomake hellomake.c hellofunc.c -I.

# Makefile - Example

```
hellomake: hellomake.c hellofunc.c  
    gcc -o hellomake hellomake.c hellofunc.c -l.
```

make with no arguments executes the first rule in the file.

By adding the list of files on which the command depends on the first line after the `;`, make knows that the rule `hellomake` needs to be executed if any of those files change.



# Makefile - Example

```
CC=gcc
```

```
CFLAGS=-I.
```

```
hellomake: hellomake.o hellofunc.o
```

```
$(CC) -o hellomake hellomake.o hellofunc.o $(CFLAGS)
```

In particular, the macro CC is the C compiler to use, and CFLAGS is the list of flags to pass to the compilation command.

By putting the object files--hellomake.o and hellofunc.o--in the dependency list and in the rule, make knows it must first compile the .c versions individually, and then build the executable hellomake. MISSING: dependency on .h file

# Makefile - Example

```
CC=gcc
```

```
CFLAGS=-I.
```

```
DEPS = hellomake.h
```

```
%.o: %.c $(DEPS)
```

```
$(CC) -c -o $@ $< $(CFLAGS)
```

```
hellomake: hellomake.o hellofunc.o
```

```
gcc -o hellomake hellomake.o hellofunc.o -l.
```

[https://www.gnu.org/software/make/manual/html\\_node/Automatic-Variables.html#Automatic-Variables](https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables)

# Makefile - Example

CC=gcc

CFLAGS=-I.

DEPS = hellomake.h

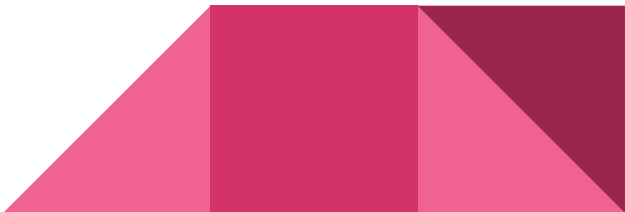
OBJ = hellomake.o hellofunc.o

%.o: %.c \$(DEPS)

\$(CC) -c -o \$@ \$< \$(CFLAGS)

hellomake: \$(OBJ)

gcc -o \$@ \$^ \$(CFLAGS)



# References


<http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>

<http://linuxide.com/linux-command/linux-diff-command-examples/>

<http://www.thegeekstuff.com/2014/12/patch-command-examples/>




# Homework

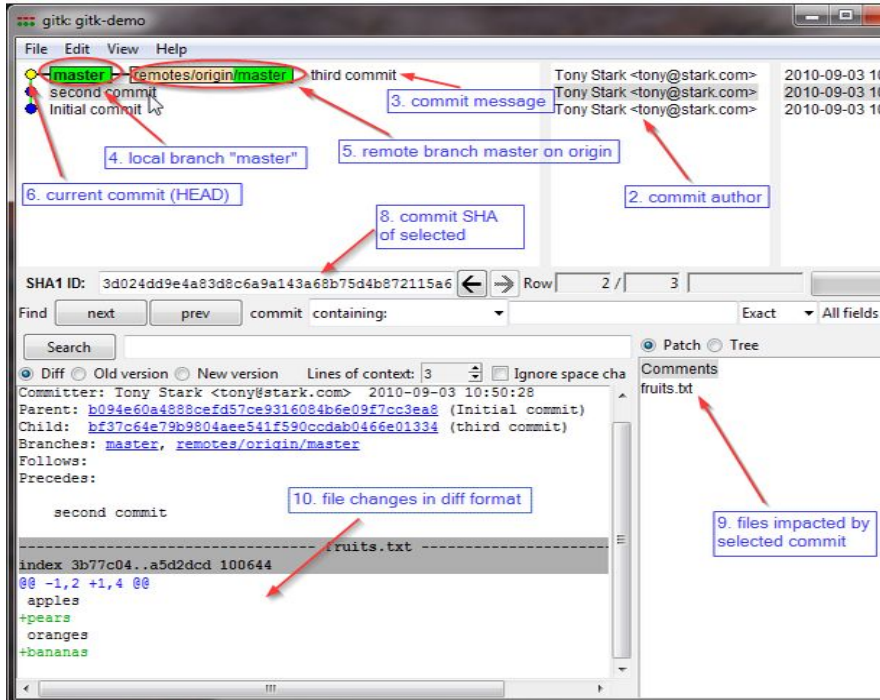
- Publish patch you made in lab 4
  - Create a new branch “quote” off of version 3.0
  - Branch command + checkout command (git branch quote v3.0; git checkout quote)
  - `$ git checkout v3.0 -b quote`
  - Use patch from lab 4 to modify this branch
  - Patch command
  - `$ patch -pnum < quote-3.0-patch.txt`
  - Modify ChangeLog-2008 file in diffutils directory
  - Add entry for your changes similar to entries in ChangeLog
- 



# Homework

- Commit changes to the new branch
  - `$ git add .`    `$ git commit -F <Changelog file>`
  - Generate a patch that other people can use to get your changes
  - `$ git format-patch -<n> --stdout > formatted-patch.txt`
  - Test your partner's patch
  - Check out version 3.0 into a tmp branch
  - Apply patch with git am command: `$ git am < formatted-patch.txt`
  - Build and test with `$ make check`
  - Make sure partner's name is in HW4.txt for #8
- 

# Gitk



- A repository browser
- Visualizes commit graphs
- Used to understand the structure of the repo
- Tutorial:  
<http://lostechies.com/joshuaflanagan/2010/09/03/use-gitk-to-understand-git/>

# Gitk

- SSH into the server with X11 enabled
  - ssh -X for OS with terminal (OS X, Linux)
  - Select “X11” option if using putty (Windows)
- Run gitk in the `~eggert/src/gnu/emacs` directory
  - Need to first update your PATH
    - `$ export PATH=/usr/local/cs/bin:$PATH`
  - Run X locally before running gitk
    - Xming on Windows

