**Lab 1:**

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

Source code is publicly available u Anyone is allowed to modify the source code u Examples u Firefox u Android u Apache

Kernel: core of OS

u Allocates time and memory to

programs

u Handles file system and communication between software and hardware

u Shell: interface between user and kernel

u Interprets commands user types in

u Takes necessary action to cause commands to be carried out

u Applications

Linux

u Linux was made by a

student called Linus

Torvalds in 1991

u Mostly Free

u Open Source

u Linux can be installed on

a variety of computer

hardware

Unix

u Unix was first developed

for multi-user and multi- tasking in mid-1970's in

BELL Labs by AT&T, GE and

MIT

u Unix is mostly used by Sun

as Solaris, HP-UX, AIX, etc

u Unix supports fewer file

systems compared to

Linux

Most important software that runs on a computer

Responsible for many things

Managing other software

Abstracting hardware

Multiuser and Multi-process Operating System

Multi-User OS- Allow many users to access/work on a single system at the same time (as long as they have their own terminal)

Multi-Process OS- Allows many processes, programs and applications to run simultaneously.

Variants :

Single User Single Task

Single User Multi Task

Multi User OS

User Interfaces: CLI vs GUI

Command Line Interface

Steep learning curve

Pure control (e.g., scripting)

Speed: Only keyboard, faster performanc

Consumes fewer resources

No change; less diverse

Graphical User Interface

More Intuitive

Limited Control

Mouse + keyboard; Slower

More resources; e.g. loading icons, fonts, etc.

Frequent changes; More diverse

Bulky remote access

Debian GNU/Linux

Clone of UNIX

Linux is just a kernel.

What is a kernel?

Core of any OS

Allocates time and memory to programs

Interfaces applications with the physical hardware

Facilitates communication between different processes (inter-process communication (IPC))

Linux distributions make the Linux kernel a completely usable OS by adding various applications

Linux distribution = GUI + GNU utilities (cp,mv,ls,etc) + installation and management tools + GNU compilers (c/c++) + Editors(vi/emacs) + ….

Shell : Interface between the user and kernel

Basics of Shell

Can be used as CLI as well as GUI depending upon the task/operation

Examples:

CLI shell on Windows :

Command Prompt

CLI shell on UNIX :

Shell

CLI on macOS :

Terminal

Basic shell commands:

<up arrow>: previous command

<tab>: auto-complete

!!: replace with previous command

![str]: refer to previous command with str

Files and Processes

Everything is either a process or a file

Process: an executing program identified by PID

File: collection of data

A document

Text of program written in high-level language

Executable

Directory

Devices

The Basics: Dealing with Files

mv: move/rename a file

cp: copy a file

rm: remove a file

r: remove directories and their contents recursively

mkdir: make a directory

rmdir: remove an empty 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

The Basics: Changing File Attributes

ln : creates a link

Hard links : Point to physical Data

Additional name for an existing file

ln file1 hlink1

Soft Links/ Symbolic Links (-s): Point to file

ln –s <source file> <my file>

touch: update access & modification time to current time

touch filename

touch -t 201101311759.30 filename

Change filename’s access & modification time to (year 2011 January day 31 time 17:59:30)

What on Earth are Hard and Soft Links?

Hard Links

An additional name for a file (.txt, .py, .doc, etc.)

If “original” is removed, the “copy” will still work

Copy is not actually a separate copy

Can’t point to directories (folders)

Soft Links (AKA symbolic links AKA symlinks)

Don’t point to the file, point to \*something\* in the file system

Can point to directories, or remote files

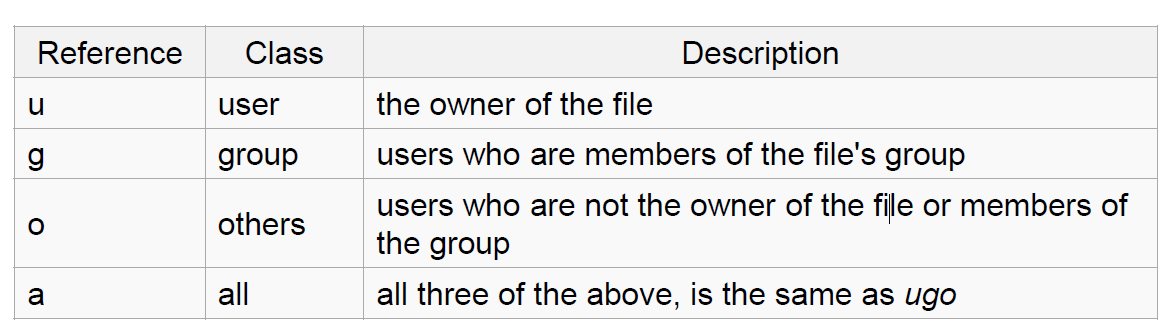
Deleting original file renders symbolic link unusable

File Permissions

chmod

read (r), write (w), executable (x)

User, group, others



Special permissions

setuid : set user ID on execution

Permits users to run certain programs with escalated privileges

E.g. : chmod u+s file1

When an executable file’s setuid permission is set, users may access the program with a level of access that matches the owner

E.g. passwd command

sticky bit (o+t)

On shared directories, it locks files within the directory from being modified/deleted by users other than the file creator, owner of the directory, or root, even if others have write permissions (Example: /tmp)

Using +t or 1

setuid, setgid (u+s, g+s)

“set user ID upon execution”

Run an executable with the permissions/priviledges of the executable’s owner or group

+s or 4,2

 For example, the setuid permission on the passwd command makes it possible for a normal user to change passwords by updating few system files like /etc/passwd and /etc/shadow which can’t be updated by non-root accounts.

Special permissions contd…

setgid : Grants permission of the group which owns the file

E.g. : chmod g+s file2

man, cat, head, tail, du, ps, kill, diff, cmp, wc, sort, echo

find command

-type: type of a file (e.g: directory, symbolic link)

-perm: permission of a file

-name: name of a file

-user: owner of a file

-maxdepth: how many levels to search

find contd…

?: matches any single character in a filename

\*: matches one or more characters in a filename

[]: matches any one of the characters between the brackets. Use ‘-’ to separate a range of consecutive characters.

Examples:

find . -name my\*

find . -name my\* -type f

find / -type f -name myfile

diff command

Examples

find . -name my\*

find . -name my\* -type f

find / -type f -name myfile –print

What will be shown:

find . –name UC\*

find . –name UC?A

Find . –iname UC\*

Process

An instance of a computer program in execution

ps

List processes that are currently running

kill

Terminate a certain process

Usage

kill PID

A process that runs in the background

Example: cron

Enables users to schedule jobs to run periodically at certain times (cron jobs)

Usage: Full Backup every month

A file comparison utility that outputs the differences between two files.

Usage:

diff file1 file2

diff –u file1 file2 (unified format)

wget command

A computer program that retrieves content from web servers

Usage

wget <URL>

dynamic value that the operating system and other software can use to determine information specific to your computer

PATH is an environment variable, specifying a set of directories where executable programs are located. In general, each executing process or user session has its own PATH setting.

whatis <command>: returns Name section of man page

whereis <command>: locates the binary, source, and manual page files for a command

which <command>: locates the executables

u A user who creates a file is also the owner and group

owner of that file.

u The file is assigned separate read, write, and execute

permissions for the owner, the group, and everyone else.

u The file owner can be changed only by the root user

u Access permissions can be changed by both the root user

and owner of the file.

u ln: create a link

u What is a link?

u It is a pointer to a file

u 2 types of links

u Soft/Symbolic Links

u Hard Links

u Symbolic Link

u Link to the original file

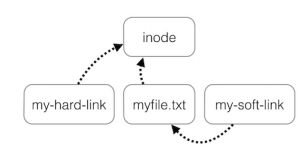
u ln –s <filename> <linkname>

u Hard Links

u Mirror copy of the original file

u Hard links: point to physical data/inode

u ln –T <filename> <linkname>



u Set of Parameters that define a user’s cultural

preferences

uLanguage

uCountry

uOther Area Specific things

u Locale command

uPrints information about the current locale

environment to standard output

u Locale gets its data from the LC\_\*environment variables

u Examples:

u LC\_TIME

u Date and time formats

u LC\_NUMERIC

u Non-monetary numeric formats

u LC\_COLLATE

u Order for comparing and sorting

u Every program has these 3

streams to interact with the

world

ustdin (0): contains data going

into a program

ustdout (1): where a program

writes its output data

ustderr (2): where a program

writes its error msg

program < file redirects file to program’s stdin:

u cat <file

u program > file redirects program's stdout to file2:

u cat <file > file2

u program 2> file redirects program's stderr to file2:

u cat < file 2> file2

u program >> file appends program’s stdout to file

u program1 | program2 assigns stdout of program1 as the

stdin of program2; text 'flows' through the pipeline

u cat <file | sort >file2

sort: sorts lines of text files

u Usage: sort [OPTION]…[FILE]…

u Sort order depends on locale

u C locale: ASCII sorting

u comm: compare two sorted files line by line

u Usage: comm [OPTION]…FILE1 FILE2

u Comparison depends on locale

u tr: translate or delete characters

u Usage: tr [OPTION]…SET1 [SET2]

u Ex: echo "12345" | tr "12" "ab

Lab 2:

-o: writes to standard output

-r: sorted in reverse order

-u: removes duplicates

echo "password a1b2c3" | tr -d [:digit:] -> password abc

echo “abc” | tr [:lower:] [:upper:] -> ABC

echo "Welcome To Lab 2" | tr [:space:] '\t’ -> Welcome To Lab 2

grep

Searches for a pattern in a file and displays all matched lines

Where is it used?

Examples:

lab2.txt

This is the second lab.

We are studying commands.

Their uses are many.

THIS LAB IS TEACHING US COMPUTER SCIENCE.

Soon this can help us do great things there.

Commands:

$grep -i ”THis" lab2.txt

$grep -c ”this" lab2.txt

$grep -w ”us" lab2.txt

$grep –e “their” –e “there” lab2.txt

Awk

define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line

Working of awk:  
 (a) Scans a file line by line  
 (b) Splits each input line into fields  
 (c) Compares input line/fields to pattern  
 (d) Performs action(s) on matched lines

Syntax:

awk options 'selection \_criteria {action }' input-file > output-file

awk variables:

NR: NR command keeps a current count of the number of input records

NF: NF command keeps a count of the number of fields within the current input record.

FS: FS command contains the field separator character which is used to divide fields on the input line. The default is “white space”.

RS: RS command stores the current record separator character.

lab2\_awk.txt

Alfa student undergrad CS

Bravo TA grad EE

Charlie student undergrad EE

Delta TA grad CS

Commands:

$ awk '{print}’ lab2\_awk.txt

$ awk ‘/student/ {print}’lab2\_awk.txt

$ awk '{print $1,$3}’ lab2\_awk.txt

$ awk '{print $0}’ lab2\_awk.txt

$ awk '{print NR,$0}’ lab2\_awk.txt

sed

Stream editor

can perform lot’s of function on file like, searching, find and replace, insertion or deletion( mainly used for substitution and find-and-replace)

Syntax:

sed OPTIONS... [SCRIPT] [INPUTFILE...]

Commands

$sed ‘s/TA/student/’ lab2.txt

$sed ‘s/TA/student/2’ lab2.txt

sed 's/\(\b[A-Z]\)/\(\1\)/g’ lab2.txt

$sed '3 s/TA/student/’ lab2.txt

$sed ‘1,3 s/TA/student/’ lab2.txt

$ sed '5d’ lab2.txt

A wildcard is a character that can stand for all members of some class of characters

The \* wildcard

The character \* is a wildcard and matches zero or more character(s) in a file (or directory) name. ( ls list\* or ls \*list)

The ? Wildcard

The character ? will match exactly one character. (ls ?list OR ls list?)

The [] Wildcard

A pair of [] represents any of the characters enclosed by them (ls \*[0-9]\*)

What is a regex: A regex is a special text string for describing a certain search pattern

Quantification

How many times of previous expression?

Most common quantifiers: ?(0 or 1), \*(0 or more), +(1 or more)

Alternation

Which choices?

Operators: [] and |

E.g Hello|World , [A B C]

Anchors

Where?

Characters: ^(beginning) and $(end)

^ start of line

$ end of line

\ turn off special meaning of next character

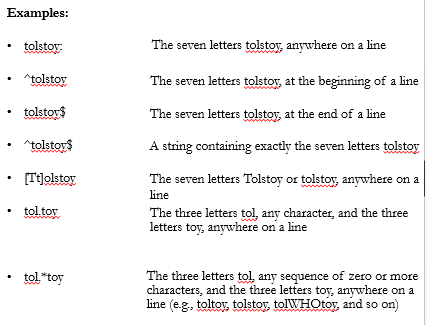
[] match any of enclosed characters, use – for range

[^ ] match any characters except those enclosed in []

. match a single character of any value

\* match 0 or more occurrences of preceding character/expression

+ match 1 or more occurrences of preceding character/expression



Basic I/O Redirection

Most programs read from stdin

Write to stdout

Send error messages to stderr

Try: $ cat // With no arguments, read standard input, write standard output

Task: Piping and Redirection

Create a file test.txt with numbers 1-5 in descending order in each line

Delete all the new line characters (with tr command and redirection) and redirect output to test1.txt

Now, first sort the file and then repeat the above step; but instead of redirection, now append the output to test1.txt

Compiled versus scripting langauges:

Compiled Languages

Examples?

C, C++,Java

First Compiled

Source code to object code; then executed

Run faster

Applications:

Typically run inside a parent program like scriptscan be compiled and used on any platform (eg. Java)

Scripting Languages

Examples?

Python, JavaScript, Shell Scripting

No compilation required. Directly interpreted!

Interpreter reads program, translates into internal form and executes

Runs slower than a high level lang

Applications:

Automation, Extracting information from a data set, Less code intensive

Shell script:

A computer program designed to be run on a shell (UNIX/Linux)

All shell commands can be executed inside a script

Why use a shell script?

Simplicity

Portability

Ease of development

Shell constructs:

Shell recognizes three fundamental kinds of commands:

Built-in commands: Commands that the shell itself executes (e.g.: echo)

Shell functions: Self-contained chunks of code, written in shell language

External Commands: mainly external utilities; backtick often associated

number=`ll | wc -l` // This is an external command

echo $number

When the shell runs a program, it asks the kernel to start a new process and run the given program in that process.

It knows how to do this for compiled programs but for a script, the kernel will fail, returning a “not executable format file” error so it’ll start a new copy of /bin/sh (the standard shell) to run the program.

But if there is more than one shell installed on the system, we need a way to tell the kernel which shell to use for a script

#! /bin/csh –f

#! /bin/awk –f

#! /bin/sh

Start with a letter or underscore and may contain any number of following letters, digits, or underscores

Hold string variables

$var1=“Hello World”

$ echo $myvar

first=rucha middle=h last=rangnekar Multiple assignments allowed on one line

fullname="$first $middle $last" Double quotes required here, for concatenating

fullname=“abc xyz mno" Use quotes for whitespace in value

oldname=$fullname Quotes not needed to preserve spaces in value

Escape Character \ - Literal value of following character

echo \|

Single Quote - Literal Meaning of all within ‘’

$hello=1

$str=‘$hello’

echo $str -> $hello

Double Quote - Literal meaning except for $, ` and \.

$hello=1

$str=“abc$hello”

echo $str -> abc1

Special Variables: certain characters reserved as special variables

$: PID of current shell

#: number of arguments the script was invoked with

n: nth argument to the script

?: exit status of the last command executed

echo $$; echo $#; echo $2; echo $?;

scalar variable vs array variable:

array\_name[index]=value; echo ${array\_name[index]}

for var in list\_values

do

command 1

..

command n

done

while condition

do

command 1

..

command n

done

for i in "${ARRAY[@]}”

do

…

Done

ALL=`ls -a $dir | sort`

declare -a ARRAY

count=0

for FILE in $ALL

do

ARRAY[$count]=$FILE

…..

Done

Conditional

if…then…fi

if…then…else…fi

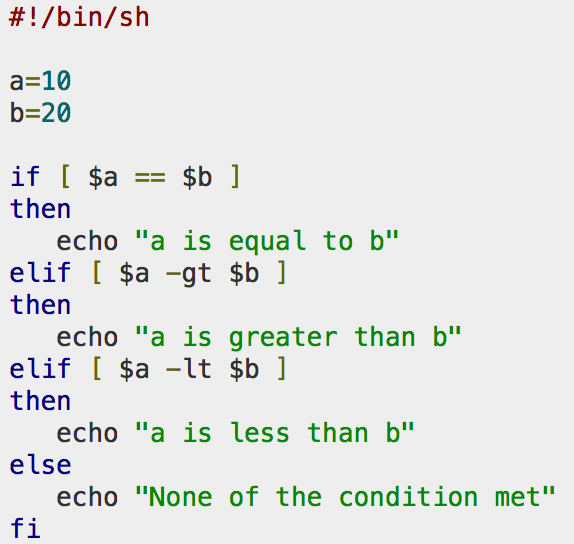
if…then…elif..then…fi

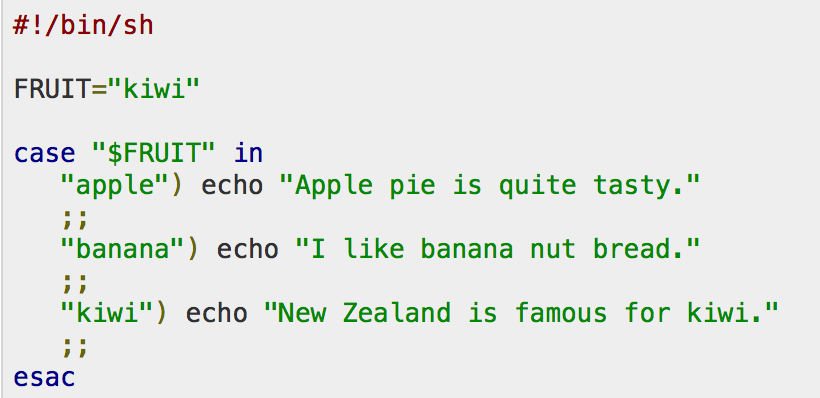
case…esac

Unconditional

break

continue





Shell prints out each command as it is executed

u Execution tracing within a script:

u set –x: to turn it on

u set +x: to turn it off

echo writes arguments to stdout, can’t output escape characters

(without –e)

u $ echo “Hello\nworld”

u Hello\nworld

u $ echo –e “Hello\nworld”

u Hello

u world

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

u $ printf "%.3e\n" 46553132.14562253

u 4.655e+07

Declared using =

u var=“hello” #NO SPACES!!!

u Referenced using $

u echo $var

u Example:

u #!/bin/sh

message="HELLO WORLD!!!"

echo $message

exit code: echo $?

Exit: Return value

Value - Typical/Conventional Meaning

u 0 - Command exited successfully.

u > 0 - Failure to execute command.

u 1-125 - Command exited unsuccessfully.

u The meanings of particular exit values are defined by each

individual command.

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

u 127 - Command not found.

u > 128 - Command died due to receiving a signal

Three kinds of quotes

u Single quotes ' '

u Do not expand at all, literal meaning

u Try temp=‘$hello$hello’ ; echo $temp

u Double quotes " "

u Almost like single quotes but expand $

u Backticks ` ` or $()

u Expand as shell commands

u Try temp=`ls` ; echo $temp

While Loop – Example:

#!/bin/sh

COUNT=6

while [ $COUNT -gt 0 ]

do

echo “Value of count is: $COUNT”

(( COUNT=COUNT-1 ))

done

Note the (( )) to do arithmetic operations

#!/bin/sh

temp=`ls`

for f in $temp

do

echo $f

done

Note: f will refer to each word in ls output

A regex is a special text string for describing a certain search pattern

u Quantification

u How many times of previous expression?

u Most common quantifiers: ?(0 or 1), \*(0 or more), +(1 or more)

u Alternation

u Which choices?

u Operators: [] and |

u E.g Hello|World , [A B C]

u Anchors

u Where?

u Characters: ^(beginning) and $(end)

Basic Regular Expressions (BRE) vs

Extended Regular Expressions (ERE)

u In basic regular expressions the meta-characters ‘?’, ‘+’,

‘{’, ‘|’, ‘(’, and ‘)’ lose their special meaning; instead use

the backslashed versions ‘\?’, ‘\+’, ‘\{’, ‘\|’, ‘\(’, and ‘\)’

for their special meaning.

u In extended regular expressions, the meta characters, ‘?’,

‘+’, ‘{’, ‘|’, ‘(’, and ‘)’ retain their special meaning. They

can be literally used by escaping them:‘\?’, ‘\+’, ‘\{’, ‘\|’,

‘\(’, and ‘\)’.

u man grep for more information

wc: outputs a one-line report of lines,

words, and bytes

u head: extract top of files

u tail: extracts bottom of files

u tr: translate or delete characters

u grep: print lines matching a pattern

u sort: sort lines of text files

u sed: filtering and transforming text

match a string that has ab followed by one or more c

Ans: abc+

match a string that has ab followed by 2 c

Ans: abc{2}

match a string that has ab followed by 2 or more c

Ans: abc{2,}

match a string that has ab followed by 2 up to 5 c

Ans: abc{2,5}

match a string that has a followed by 2 up to 5 copies of the sequence bc

Ans: a(bc){2,5}

match a string that has a followed by b or c

Ans: a(b|c)

Ans: a[bc]

Match a string that has a character from 0 to 9 before a % sign

Ans: [0-9]%

Match a string that does not have a letter from a to z or from A to Z

Ans: [^a-zA-Z]

match a d only if is followed by r, but r will not be part of the overall regex match

Ans: d(?=r)

match a d only if is preceded by an r, but r will not be part of the overall regex match

Ans: (?<=r)d

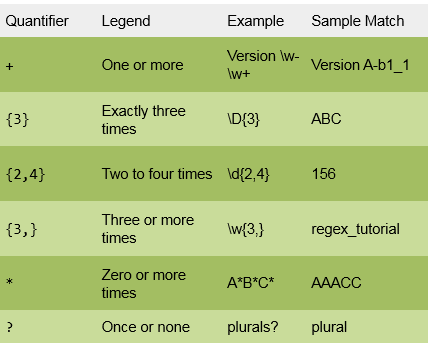
Difference between [.] and (.)

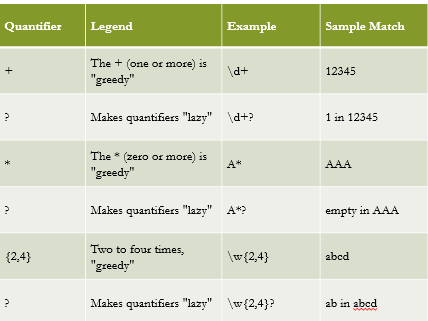
The [.] will match exactly the ‘.’ character.

eg: .

The (.) will match will match any one character.

eg: a, b, 1





**Input String:**

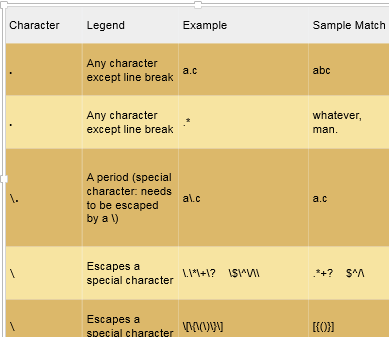
**<em>Hello World</em>**

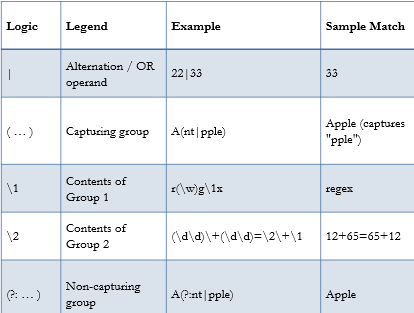
**Regex 1:**

**<.+>**

**Regex 2:**

**(<.+?>)**





**Lab #3**

“Python is a programming language that lets you work quickly and integrate systems more effectively.” - python.org

Friendly

Relatively straightforward code

Easy to get going

Flexible

Machine learning (TensorFlow, PyTorch, Caffe, scikit-learn)

Web (Django, Flask, Tornado)

Almost anything you might want to do

Object oriented

Not the fastest, but its ease of use makes up for it

Can be used via interactive shell

Also called REPL (read-eval-print loop)

Just type “python” or “python3” into your shell

Can be called to invoke a program

python3 ./train.py

Can make standalone scripts using shebang

./python\_poornames

#!/usr/bin/env python3

counter = 100 # An integer assignment

miles = 1000.0 # A floating point

name = "John" # A string

print(counter)

print(miles)

print(name)

print(5/2) # 2.5

Python 2 versus 3:

raw\_input() gone, replaced with input()

Ints behave differently

Strings/bytes a bit different

Exceptions can’t be unpacked

Lots more

Even that isn’t everything

Conditonals:

x = 5

y = 8

if x == 7:

print(“The value of X was {}”.format(x))

if x <= 7:

print(“hello”)

elif y <= 7:

print (“x fails, y good”)

else:

print(“Not good”)

if (x <=7) and (y == 8):

print(“Condition valid”)

Lists:

Like arrays, but not

Can change their size

Can hold items of different types

x = [1, 1.5, “Goodbye”]

y = []

z = x + x

x[3] # Error!

x[-1] # “Goodbye”

x.append(y)

x.append(x)

x[0] # 1

Allows for creative access into lists + strings

Format: my\_list[start:stop:step]

Alternate format: my\_list[start:stop]

x = “hello string”

x[1:] # “ello string”

x[:-1] # “hello strin”

x[0::2] # “hlosrn”

x[::-1] # “gnirts olleh”

An iterable is anything you can iterate over

Can call next(X) on it

for x in iterable:

print(x)

Dictionaries:

Map keys to values

Strings -> lists

Ints -> floats

Etc.

Keys must be unique

Support data types

X = {} # Empty dict

Y = {1: “hi”, “I”: 3}

X[“bye”] = 18

Y[1] # “hi”

Opening files:

with open(filename) as my\_file:

for line in my\_file:

print(line)

def cool\_math(a, b, c=12, d=15):

return a + b + c + d

def cool\_math\_two(a, b):

return a + b

cool\_math(5, 6) # 38

cool\_math(5, 6, d=0) # 23

cool\_math\_two(5, 6) # 11

cool\_math(“ab”, “cd”) # “abcd”

A way to report type of problem + what went wrong

Allows new code paths when errors are encountered

Try/Except let you handle them

try:

x = int(input("Please enter a number: "))

break

except ValueError as e:

print("Oops! That wasn’t a number!")

if True:

raise ValueError(“uh oh!”

Let you keep state

Let you reuse code

Help you organize code

class Rectangle:

def \_\_init\_\_(self, x, y):

self.l = x

self.b = y

def getArea(self):

return self.l \* self.b

def getPerimeter(self):

return 2 \* (self.l + self.b)

def main():

rect = Rectangle(3, 4)

print("Area of Rectangle:", rect.getArea())

print("Perimeter:", rect.getPerimeter())

main()

Unpacking:

def useless():

return (5, 6)

A, b = useless()

Installign SW:

Linux

– rpm(Redhat Package Management)

• RedHat Linux (.rpm)

– apt-get(Advanced Package Tool)

• Debian Linux, Ubuntu Linux (.deb)

– Good old build process

• configure, make, make install

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

Compilation process:

Prog1.cpp goes thorugh C++ preprocessor, creates expanded temporary file, which is compiled by compiler into .s assembler file, then assembled into .o object code file by assembler, and then linked with other .o and .so files by a linker to create an exe

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

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

Solution: use makefiles

Utility for managing large software projects

• Compiles files and keeps them up-to-date

• Efficient Compilation (only files that need to

be recompiled)

Makefile Example

# Makefile - A Basic Example

all : shop #usually first

shop : item.o shoppingList.o shop.o

g++ -g -Wall -o shop item.o shoppingList.o shop.o

item.o : item.cpp item.h

g++ -g -Wall -c item.cpp

shoppingList.o : shoppingList.cpp shoppingList.h

g++ -g -Wall -c shoppingList.cpp

shop.o : shop.cpp item.h shoppingList.h

g++ -g -Wall -c shop.cpp

clean :

rm -f item.o shoppingList.o shop.o shop

Build Process

• configure

– 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)

Phony targets:

A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request. There are two reasons to use a phony target: to avoid a conflict with a file of the same name, and to improve performance.

If you write a rule whose recipe will not create the target file, the recipe will be executed every time the target comes up for remaking. Here is an example:

clean:

rm \*.o temp

Because the rm command does not create a file named clean, probably no such file will ever exist. Therefore, the rm command will be executed every time you say ‘make clean’.

In this example, the clean target will not work properly if a file named clean is ever created in this directory. Since it has no prerequisites, clean would always be considered up to date and its recipe would not be executed. To avoid this problem you can explicitly declare the target to be phony by making it a prerequisite of the special target .PHONY (see Special Built-in Target Names) as follows:

.PHONY: clean

clean:

rm \*.o temp

Once this is done, ‘make clean’ will run the recipe regardless of whether there is a file named clean.

Not just a scripting language

• Object-Oriented language

– Classes

– Member functions

• Compiled and interpreted

– Python code is compiled to bytecode

– Bytecode interpreted by Python interpreter

• Not as fast as C but easy to learn, read and use

• Very popular at Google and other big companies

Why is it popular?

• Uses English keywords frequently where

other use different punctuation symbols

• Fewer Syntactical Constructions

• Automatic Garbage Collection

• Easy integration with other programming

Languages

Different Modes

• Interactive:

• Run commands on the python shell without actually

writing a script/program.

• Script Mode:

• Type a set of commands into a script

• Execute all the commands at once by running the

script

Common data structure in Python

• A python list is like a C array but much more:

– Dynamic (mutable): expands as new items are

added

– Heterogeneous: can hold objects of different types

• How to access elements?

– List\_name[index]

for i in range(len(list1)):

print i

The raw\_input([prompt]) function reads one line from

standard input and returns it as a string (removing the trailing

newline)

• s = raw\_input("Enter your input: ");

• print("Received input is : ", s)

The input([prompt]) function is equivalent to raw\_input,

except that it assumes the input is a valid Python expression

and returns the evaluated result to you.

• str = input("Enter your input: ");

• print(“Received input is : ", str)

Why Build From Source?

Configure for your system, Apply your own tweaks, Developing your own programs

Patching

• 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

diff Unified Format

• diff –u original\_file modified\_file

• --- path/to/original\_file

• +++ path/to/modified\_file

• @@ -l,s +l,s @@

– @@: beginning of a hunk

– l: beginning line number

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

– A line with a:

• - sign was deleted from the original

• + sign was added to the original

• stayed the same

**Lab #4:**

Using GDB

Compile Program

Normally: $ gcc [flags] <source files> -o <output file>

Debugging: $ gcc [other flags] –g <source files> -o <output file>

enables built-in debugging support

pass -O0 to turn off optimizations, which can make debugging difficult

Specify Program to Debug

$ gdb <executable>

or

$ gdb

(gdb) file <executable>

Using GDB

3. Run Program

(gdb) run or

(gdb) run [arguments]

4. In GDB Interactive Shell

Tab to Autocomplete, up-down arrows to recall history

help [command] to get more info about a command

5. Exit the gdb Debugger

(gdb) quit

Runtime Errors

Segmentation faults

Normally just says “segmentation fault”

If run in a debugger, we instead get the following, which lets us see exactly where and why we crashed

\* thread #1, queue = 'com.apple.main-thread', stop reason = EXC\_BAD\_ACCESS (code=1, address=0x0)

frame #0: 0x0000000100000f48 hello`main at hello.c:10:10

7 }

8 int secret\_value = my\_val;

9 int \*bro;

-> 10 \*bro = 208;

11 printf("This is just a message: %d\n", secret\_value);

12 return 0;

13 }

Target 0: (hello) stopped.

Breakpoints

Breakpoints

used to stop the running program at a specific point

If the program reaches that location when running, it will pause and prompt you for another command

You can set as many as you want

Example:

(gdb) break file1.c:6

Program will pause when it reaches line 6 of file1.c

(gdb) break my\_function

Program will pause at the first line of my\_function every time it is called

(gdb) break [position] if expression

Program will pause at specified position only when the expression evaluates to true

expression can be anything valid in the language you’re debugging

Managing Breakpoints

(gdb) delete [bp\_number | range]

Deletes the specified breakpoint or range of breakpoints

(gdb) disable [ bp\_number | range]

Temporarily deactivates a breakpoint or a range of breakpoints

(gdb) enable [ bp\_number | range]

Restores disabled breakpoints

If no arguments are provided to the above commands, all breakpoints are affected!!

(gdb) ignore bp\_number iterations

Instructs GDB to pass over a breakpoint without stopping a certain number of times.

bp\_number: the number of a breakpoint

Iterations: the number of times you want it to be passed over

Displaying Data

You’ve paused execution with a breakpoint, now you want to look inside the program

(gdb) print [/format] expression

prints the value of the specified expression in the specified format

Formats:

d: Decimal notation (default format for integers)

x: Hexadecimal

o: Octal

t: binary

Continuing Execution

Four ways to continue execution after a breakpoint

c or continue: debugger will continue executing until next breakpoint

s or step: debugger will continue to next source line

n or next: debugger will continue to next source line in the current (innermost) stack frame

f or finish: debugger will resume execution until the current function returns. Execution stops immediately after the program flow returns to the function's caller

the function's return value and the line containing the next statement are displayed

Watchpoints

Watch/observe changes to variables

(gdb) watch my\_var

sets a watchpoint on my\_var

the debugger will stop the program whenever the value of my\_var changes

prints out new and old values

(gdb) rwatch expression

Execution stops whenever the expression is read

Process Memory Layout

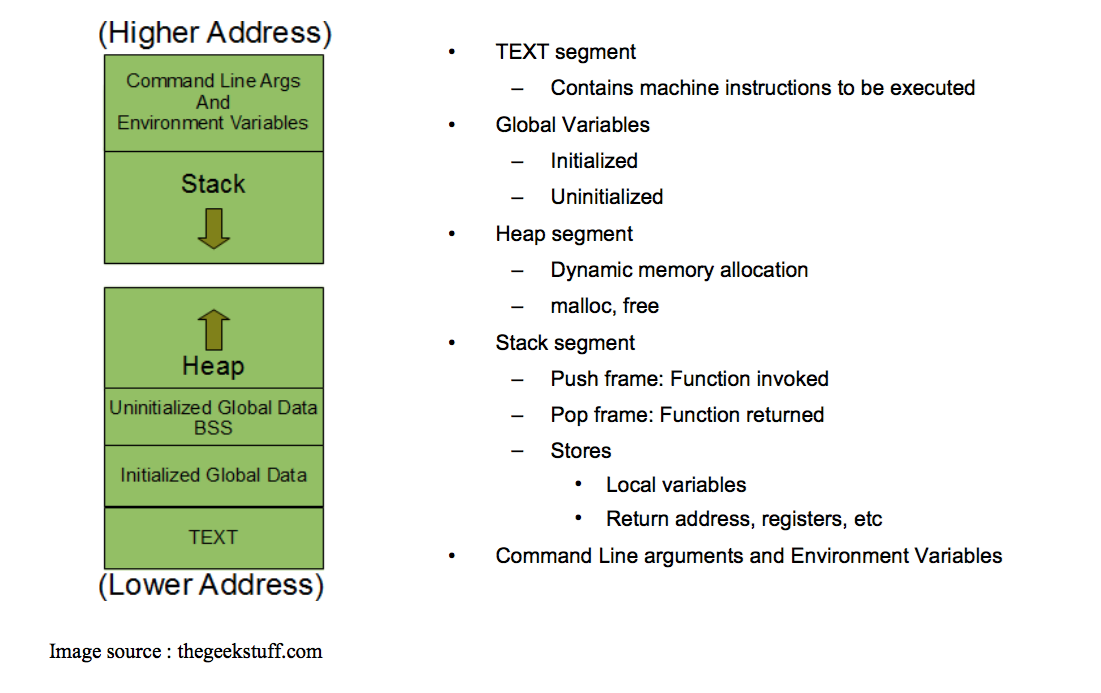
Processes think they own all the memory

Done via “Virtual Memory”

Some systems, such as microcontrollers, might not use this

Exact calling conventions (registers vs stack, etc.) can vary across architecture and OS

Blog post explaining this in x86 (non 64 bit)



Process Memory Layout

There’s a ton of abstraction going on

Don’t worry too much about what’s physically happening, that’s for another class

Caching

Write back / write through

Cache policies

Cache coherence on multicore / multichip machines

Virtual Memory

Paging

Stack Frames

Every time a function is called, some memory is set aside for it

This memory is called a stack frame

Stack frames consist of the following

Storage space for all local variables

Address of next line of code to execute after the function

The arguments to the function

Analyzing the Stack in GDB

(gdb) backtrace|bt

Shows the call trace (the call stack)

Without function calls:

#0 main () at program.c:10

one frame on the stack, numbered 0, and it belongs to main()

After call to function display()

#0 display (z=5, zptr=0xbffffb34) at program.c:15

#1 0x08048455 in main () at program.c:10

Two stack frames: frame 1 belonging to main() and frame 0 belonging to display().

Each frame listing gives

the arguments to that function

the line number that's currently being executed within that frame

Analyzing the Stack

(gdb) info frame

Displays information about the current stack frame, including its return address and saved register values

(gdb) info locals

Lists the local variables of the function corresponding to the stack frame, with their current values

(gdb) info args

List the argument values of the corresponding function call

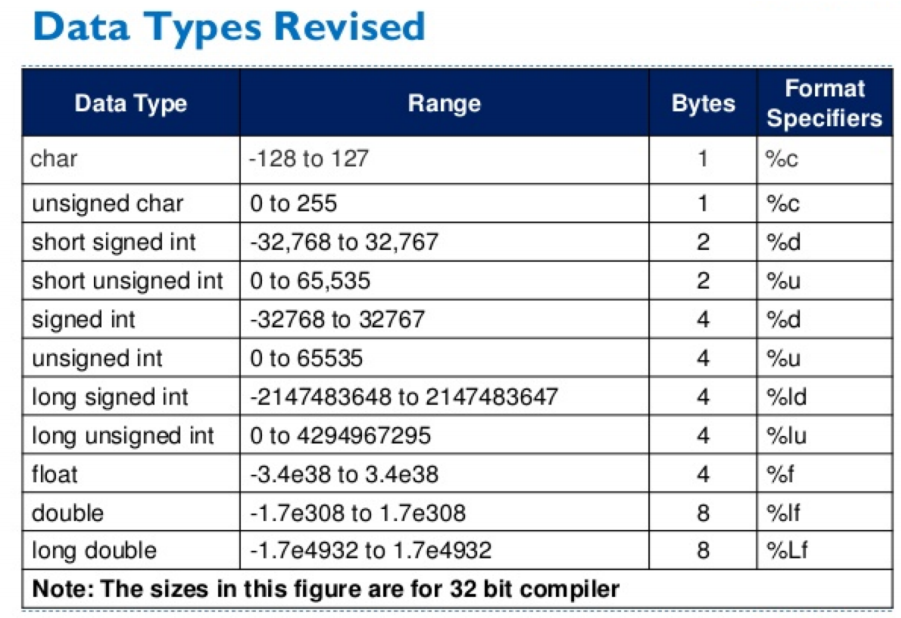
Other Useful Commands

(gdb) info functions

Lists all functions in the program

(gdb) list

Lists source code lines around the current line



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 Student {

char name[64];

char UID[10];

int age;

int year;

} student;

student s;

Normal variable stores the value whereas pointer variable stores the address of

the variable

u The content of the C pointer always be a whole number i.e. address

u C pointer is always initialized to null, i.e. int \*p = null

u The value of null pointer is 0

u & symbol is used to get the address of the variable

u \* symbol is used to get the value of the variable that the pointer is pointing to

u If a pointer in C is assigned to NULL, it means it is pointing to nothing

u Two pointers can be subtracted to know how many elements are available

between these two pointers

u But pointer addition, multiplication, division are not allowed

u The size of any pointer is 4 byte (for 32 bit compiler).

Accessing the value that the pointer points to

u Example:

u double x, \*ptr;

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

u \*ptr = 7.8; // assign the value 7.8 to x

char c = ‘A’ char \*cPtr = &c char \*\*cPtrPtr = &cPtr

Dynamic Memory

u Memory that is allocated at runtime

u Allocated on the heap

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

u Allocates size bytes and returns a pointer to the allocated memory

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

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

bytes

u void free (void \*ptr);

u Frees the block of memory pointed to by ptr

Pointers to functions:

Also known as: function pointers

Goal: write a sorting function

Has to work for normal reverse sorting + anything else you dream of

How?

Write multiple functions

Provide a flag as an argument to the function

Polymorphism and virtual functions

Use function pointers!!

Pointers to Functions

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 );

qsort Example

void qsort (void\* base, size\_t num, size\_t size, int (\*compar)(const void\*,const void\*));

Return value meaning for comparator function:

< 0 The element pointed by p1 goes before the element pointed by p2

= 0 The element pointed by p1 is equivalent to the element pointed by p2

> 0 The element pointed by p1 goes after the element pointed by p2

#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;

}

Reading/Writing Characters

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[] = “John”;

fp = fopen(“file.txt”, “w+”)

fprintf(fp, “%s has %d points.\n”, player, score);

**Lab 5:**

Operating Systems (again)

Your program thinks it owns all memory

Your program can’t read the memory of others

Your program can somehow talk to hardware??

Your laptop can have multiple programs running at once

Keeps running even if one crashes

How?

The operating system

More specifically: its kernel

Kernel

The kernel is the core of the OS

It’s a program too!

Interface between hardware and software

Controls access to system resources: memory, I/O, CPU

Manages CPU resources, memory resources, processes

Lowest layer above the CPU

Ensure protection and fair allocations

Which Code is Trusted?

The Kernel ONLY

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

CPU Modes

Modes that place restrictions on what the running process can do

User mode: restricted access to system resources

Kernel mode: Unrestricted access

CPU Modes

Hardware contains a mode bit

e.g. 0 for kernel mode, 1 for user mode

Varies by architecture

x86 has 4 privilege levels

We’ll stick to two

User mode

Certain instructions unavailable

Can only interact with subset of memory

Kernel mode

All instructions available

Can interact with any memory it wants

User space vs Kernel Space

Virtual memory means that memory addresses are faked

Memory split into two areas

User space

Kernel space

User space

Where user programs live + run

Keeps user programs “in the dark”

Kernel space

Where kernel runs

Holds code for kernel

We only trust this kernel code!

What About User Processes?

The kernel executes privileged operations on behalf of untrusted user processes

Kernel Design Philosophies

Monolithic kernels

The entire operating system runs in kernel space

Approach used by Linux (classically), most UNIX style kernels

Microkernels

As much functionality is moved out of the kernel as possible

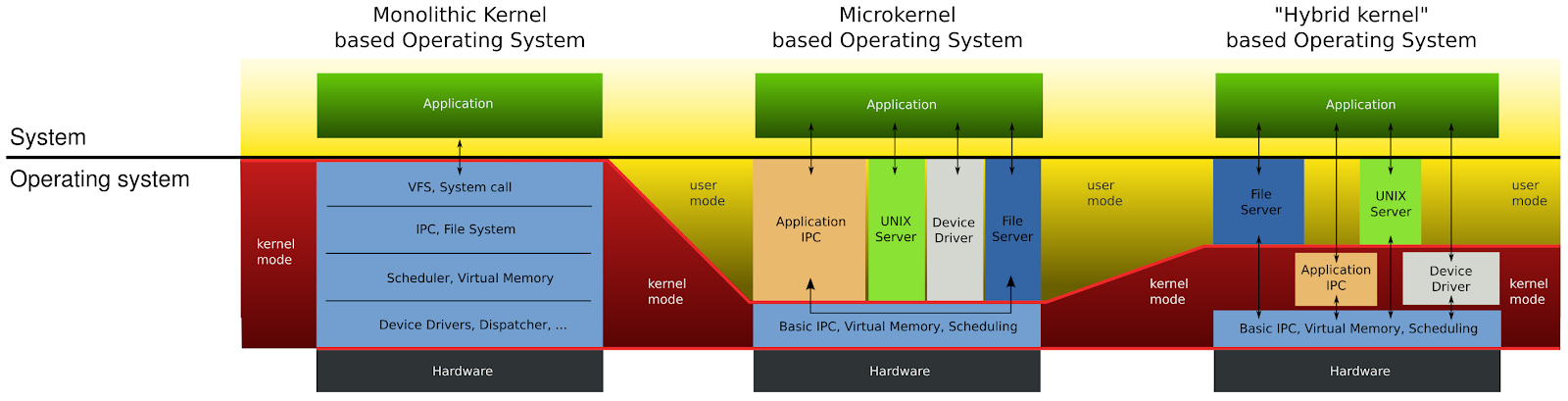
Even traditional OS components like file systems + networking

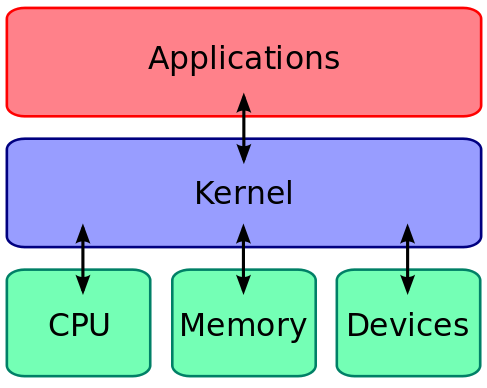
Easier to maintain

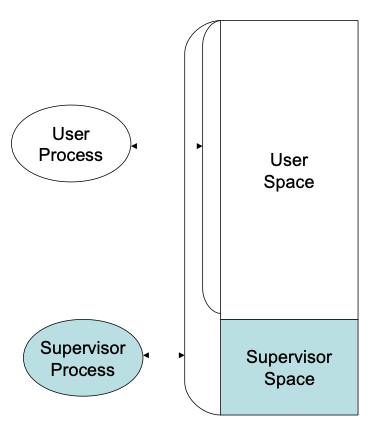
Hybrid kernels

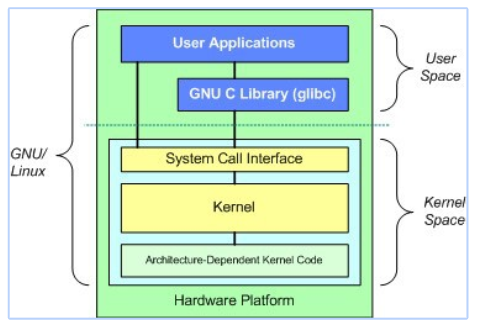
Used in Windows, XNU

Like microkernels, but with some functionality pulled back into kernel for performance









System Calls

Special type of function that:

Provide interface between user programs and OS

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

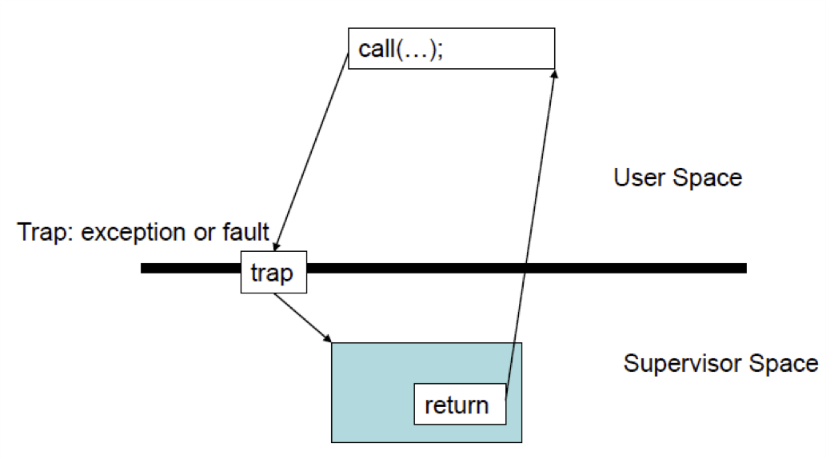
Verifies that the user should be allowed to do the requested action and then does the action (kernel performs the operation on behalf of the user)

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

Making a System Call

System calls are directly available and used in high level languages like C and C++

Hence, easy to use system calls in programs

For a programmer, system calls are same as calling a procedure or function

So, what is the difference between a system call and a normal function?

System call enters a kernel

Normal function does not and cannot enter a kernel!

Making a System Call

App developers might not have direct access to system calls sometimes

They have to invoke the API

The functions in the API invoke the actual system calls

Advantages:

Portability: as long as a system supports an API, any program using that API can compile and run

Ease of Use: using API is significantly easier than the actual system call

Types of System Calls

5 categories:

Process Control

A running program needs to be able to stop execution

Normally or abnormally

If abnormally, dump of memory is created and taken for examination by a debugger

File Management

To perform operations on files

Create, delete, read, write, reposition, close

Many a times, OS provides an API to make these system calls

Device Management

Process usually requires several resources to execute

If available, access granted

Resources = devices

Eg: physical I/O devices attached

Information Management

To transfer information between user program and OS

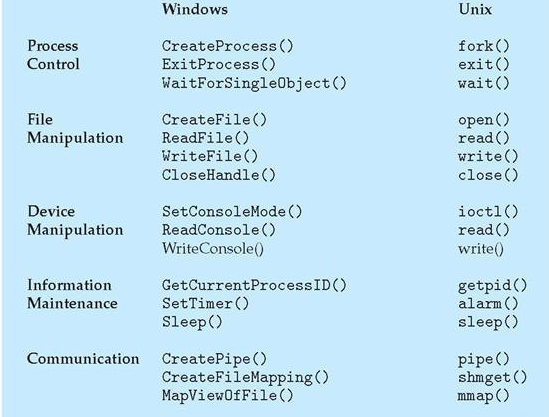
Eg: time, date

Communication

Interprocess communication

Message passing model

Shared memory model



What if there were no System Calls?

Kernel can be accessed by anyone!

Threat to the security of OS

Example System Calls

#include <fcntl.h>

#include <sys/stat.h>

#include <unistd.h>

int open(const char \*pathname,

int close(int fd);

File descriptors

int flags, mode\_t

mode);

0 stdin, 1 stdout, 2 stderr

ssize\_t read(int fildes, void \*buf, size\_t nbyte)

fildes: file descriptor buf: buffer to write to

nbyte: number of bytes to read

ssize\_t write(int fildes, const void

\*buf, size\_t nbyte);

fildes: file descriptor

buf: buffer to write from

nbyte: number of bytes to write

#include <sys/stat.h>

int fstat(int filedes, struct stat \*buf)

Returns information about the file with the descriptor filedes into buf

Library Functions

Functions that are a part of standard C library

To avoid system call overhead use equivalent library functions

getchar, putchar vs. read, write (for standard I/O)

fopen, fclose vs. open, close (for file I/O), etc.

How do these functions perform privileged operations?

They make system calls

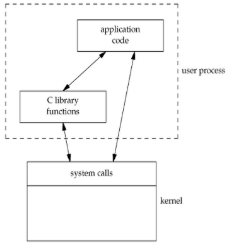
So What’s the Point?

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

These are the two modes in which a program executes

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

u User mode

u CPU restricted to unprivileged instructions and a specified area of memory

u Less privileged

u Exception will crash single process

u Supervisor/kernel mode

u CPU is unrestricted, can use all instructions, access all areas of memory and take

over the CPU anytime

u High privilege

u Exception will crash the entire OS

User Space vs. Kernel Space

u RAM is divided into distinct regions

u User space - where normal user processes run

u limited access to system resources: memory, I/O, CPU

u user mode applications are run and memory is swapped out for

different applications

u Kernel space

u stores the code of the kernel, which manages processes

u prevent processes messing with each other and the machine

u strictly reserved for running the kernel (Background processes, OS and

most device drivers)

File Descriptors

File descriptor is an integer that uniquely identifies an open file of the process

File descriptor table is the collection of integer array indices that are file descriptors in which elements are pointers to file table entries. One unique file descriptors table is provided in operating system for each process

Read from stdin => read from fd 0: Whenever we write any character from keyboard, it is read from stdin through fd 0

Write to stdout => write to fd 1: Whenever we see any output to the video screen it is written to stdout in screen through fd 1.

Write to stderr => write to fd 2: Whenever we see any error to the video screen, it is written to stderr in screen through fd 2.

Open()

Used to Open the file for reading, writing or both

Syntax: int open (const char\* Path, int flags);

Path: Path to file which is to be opened

Use Relative path if you are working in the same working directory as file

Otherwise, Absolute path, starting with ‘/’

Flags

O\_RDONLY: read only,

O\_WRONLY: write only,

O\_RDWR: read and write,

O\_CREAT: create file if it doesn’t exist,

O\_EXCL: prevent creation if it already exists

Returns a file descriptor

Create()

Used to create a new empty file

Syntax: int create(char \*filename, mode\_t mode)

Filename: name of the file which you want to create

Mode: Indicates permission of the new file

returns first unused file descriptor (generally 3 because 0, 1, 2 fd are reserved) ; returns -1 when error

Difference between open() and create()

create function creates files, but can not open existing file.

Create is more of a legacy function now

creat() is equivalent to open() with flags equal to O\_CREAT|O\_WRONLY|O\_TRUNC

Close()

Closes the file which pointed by fd

Frees the file descriptor

Syntax: int close(int fd);

Fd: File Descriptor

Returns 0 on success and -1 on error

Read()

From the file indicated by the file descriptor fd, the read() function reads n bytes of input into the memory area indicated by buf.

Syntax: size\_t read (int fd, void\* buf, size\_t n);

fd: file descriptor

buf: buffer to read data from

n: length of buffer

Returns:

Number of bytes read on success

File position advanced by this many bytes

0 on reaching end of file

-1 on error

Write()

Writes n bytes from buf to the file associated with fd

Syntax: size\_t write (int fd, void\* buf, size\_t cnt);

fd: file descriptor

buf: buffer to write data to

n: length of buffer

Returns Number of bytes written on success

0 written and fd is a regular file? Might return an error via errno (in a few slides)

Maybe no error will be detected

0 written and fd isn’t regular? Results aren’t specified

-1 on error

Syscalls: Checking what went wrong

Many syscalls will return an error value (often -1)

But how do you know what the error was?

Syscalls will often set the value of errno on failure

Use errno like a variable

Can check the value of errno to check the exact error and handle it

EBADF Bad file descriptor

EDQUOT Disk quota exceeded

EACCES Permission denied

etc.

errno

#include <stdio.h>

int main(void)

{

FILE \*f = fopen("non\_existent", "r");

if (f == NULL) {

perror("fopen() failed");

} else {

fclose(f);

}

}

Output: fopen() failed: No such file or directory

strerror will give you a char\* of the error message

errno

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

int main(void)

{

int result = close(5);

if (result == -1) {

// We saw an error

if (errno == EINTR) {

printf(“Interrupted!\n”);

}

}

}

Why use library functions?

Portability

Don’t need to rewrite your program when hopping platforms

Speed

Library may use tricks (buffered I/O) to make things faster

Tr2b.c

Write a main function which accepts arguments

main(int argc, const char\* argv[])

Check for the length of arguments

Store first argument as char \* from, second argument as char \* to

Compare the lengths of from and to; If not the same, throw an error and exit

You can use strlen to get the length of strings

To throw an error, write to stderr using library functions

Check if ‘from’ has duplicates

If so, throw an error

Take input from stdin (till you reach eof of stdin) using getchar()

Check if the character you just retrieved is a part of from

If yes, put the corresponding character in stdout with putchar()

Otherwise output original character

Tr2u.c

Repeat the same procedure as in tr2b.c except replace:

getchar() with read

putchar() with write

Time and strace

time [options] command [arguments...]

Can give a very broad picture of performance

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 (in syscalls)

strace: intercepts and prints out system calls.

–$ strace –c ./tr2b ‘AB’ ‘XY’ < input.txt

–$ strace –c ./tr2u ‘AB’ ‘XY’ < input.txt

Assignment 5 - Homework

Rewrite sfrob using system calls in place of stdio (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

Accepts a -f option, which specifies case insensitive sorting via toupper

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

fstat() demo

Do “man 2 stat” in shell for additional information

Opens the syscall section of man

Check if it is a regular file or piped input through S\_ISREG

Use lseek() in case file has grown in size and set the file offset to the current location

fsanitize

Don’t ignore this slide, it’s very helpful

Checks for memory issues (address mode) and runtime exceptions (undefined mode)

Make sure to have /usr/local/cs/bin/ prepended to the path (works with the latest version of gcc)

Compile using

gcc –o out program-with-potential-memory-leak.c –fsanitize=address –static-libasan

OR

gcc –o out program-with-potential-undefined-behavior.c –fsanitize=undefined –static-libubsan

Can use both at once if you want

Assignment 5 - Homework

Refer to Read, Write, Open, Close System Calls

Reserved File Descriptors

0 – stdin

1 – stdout

2 – stderr

int fstat(int fd, struct stat \*buf)

Returns information about the file with the descriptor fd into buf

**Lab #6:**

What happens between us writing our C program and it executing?

CALL

Compiler

Takes in source, outputs assembly (still human readable)

Assembler

Takes in assembly, outputs machine code

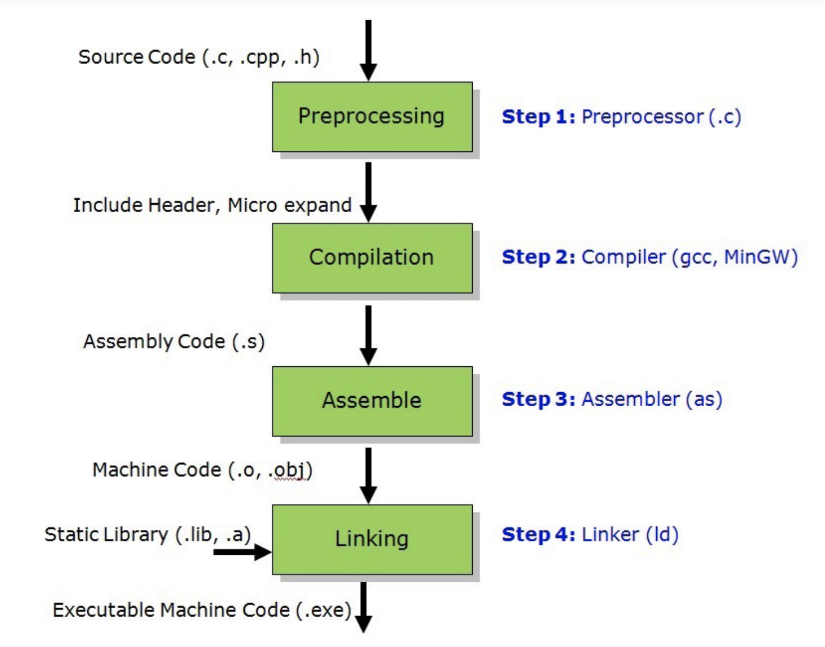
Linker

Fills in the blanks, allowing program to use libraries

Loader

Does different things on different platforms

Takes program from a file on disk to an executing process



Static Linking

Carried out only once

If static libraries are used, the linker will copy all the modules referenced by the program into the executable

Static libraries typically use a “.a” file extension

Dynamic Linking

Allows a process to add, remove, replace or

relocate object modules during its execution.

If shared (dynamic) libraries are used:

Only copy a little reference information when the executable file is created

Complete the linking during loading time or running time

Dynamic libraries typically use a “.so” file extension

.dll on Windows

Linking and Loading

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

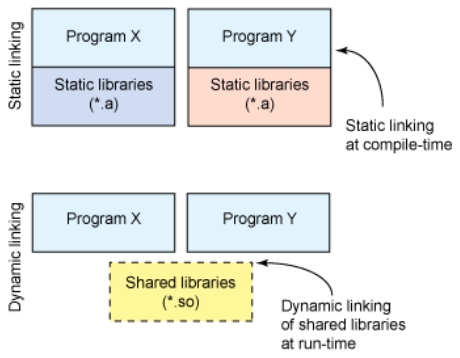
Why isn't everything written as just one big program, saving the necessity of linking?

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

Use a new library without recompiling / redistributing

Avoid reading absolutely everything into memory

Allow programs to share the memory used by the library



Dynamic linking

Unix systems: Code is typically compiled as a dynamic shared object (DSO)

Dynamic vs. static linking resulting size

$ gcc -static hello.c -o hello-static

$ gcc hello.c -o hello-dynamic

$ ls -l hello

80 hello.c

13724 hello-dynamic

1688756 hello-static

Pros and cons?

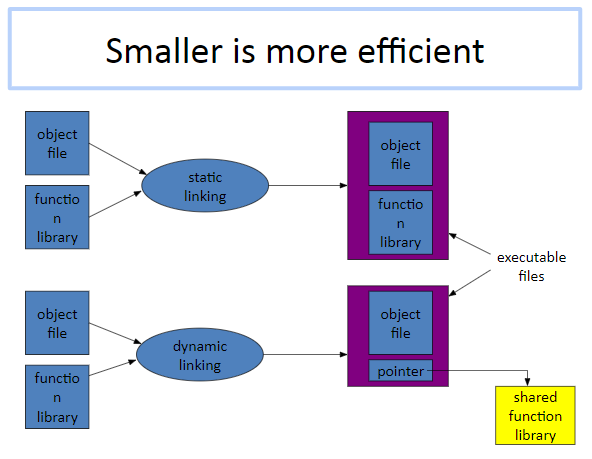
Advantages of dynamic linking

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



Disadvantages of dynamic linking

(Slight) performance hit

Need to load shared objects (at least once)

Need to resolve addresses (once or every time)

Remember back to the system call assignment…

What if the necessary dynamic library is missing?

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

ldd

Usage: ldd ./my\_program

Prints out the shared libraries used by a program

[tylerd@lnxsrv07 ~]$ ldd ./sfrob

linux-vdso.so.1 => (0x00007ffe749ec000)

librt.so.1 => /lib64/librt.so.1 (0x00007fe70f395000)

libdl.so.2 => /lib64/libdl.so.2 (0x00007fe70f191000)

libpthread.so.0 => /lib64/libpthread.so.0 (0x00007fe70ef75000)

libm.so.6 => /lib64/libm.so.6 (0x00007fe70ec73000)

libgcc\_s.so.1 => /lib64/libgcc\_s.so.1 (0x00007fe70ea5d000)

libc.so.6 => /lib64/libc.so.6 (0x00007fe70e68f000)

/lib64/ld-linux-x86-64.so.2 (0x00007fe70f59d000)

Lab 6

Build the code simpgmp.c

Use ldd to investigate which dynamic libraries your program loads

Use strace to investigate which system calls your program makes

Use “ls /usr/bin | awk ‘NR%101==nnnnnnnnn%101’” to find ~12 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 omit the duplicates!

Program Library Anatomy

Static libraries: installed into program before the it is run

Dynamic linked libraries: loaded at program start-up

Dynamic loaded libraries: loaded and used at any time while a program is running

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 (shared object) file extension in Unix system

.dll (dynamically linked library) on Windows

Why we need shared libraries?

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

Pros and Cons for Dynamic Linking

+ The executable is typically smaller

+ More flexible for changes

+ Concurrency

- Performance

- Missing libraries

- Wrong library version

Dynamic Loading

A mechanism to load shared library to memory at runtime

E.g. in a C program, use a line of code to load a new library when that line of code is being executed

You still need to load a shared library

What’re the advantages?

Allow start up in the absence of some library

Avoid linking unnecessary libraries

Big Picture

1. Create a static/shared library

Use ar (for static) or gcc (for dynamic)

2. Link that library when you compile your program

Use gcc (note all the flags)

3. If necessary, use dynamic load in your program

dlopen(), dlsym(), dlclose()

4. Automate the process

make

Create Libraries

Compile the files with -c (means without the linking step) flag. If you want to create a dynamic library, add -fPIC flag.

$ gcc (-fPIC) -c [file.c] -o [file.o]

To create a static library, use

$ ar rcs [libxxx.a] file.o

To create a shared library, use

$ gcc -shared [file.o] [libxxx.so]

Link Libraries

To link the library statically/dynamically

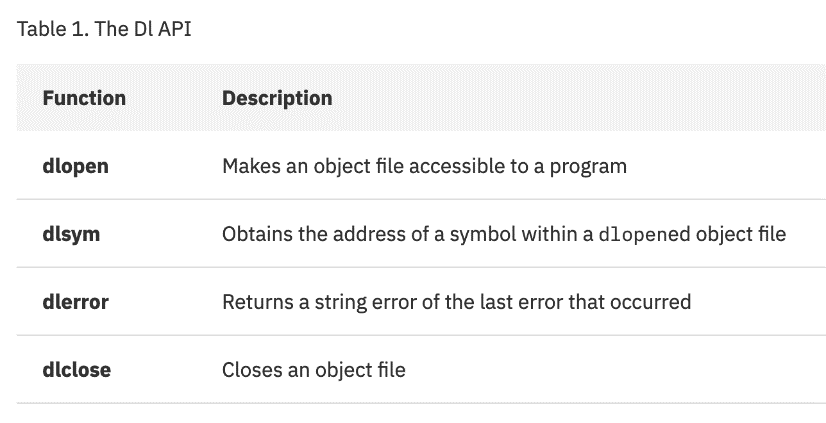
$ gcc -L[path] -l[xxx] [your\_main.o] -o [your\_main]

where [your\_main.o] is the object file (you previously compiled without linking) that depends on your library, [path] is the location of the library, and [xxx] is the name of the library without prefix “lib”

If library name is libtest.so, here you should use -ltest

To make sure the linker can find your library, either add your directory path to $LD\_LIBRARY\_PATH, or use -ldl -Wl,-rpath=[path] when compiling (pay attention to the space)

-Wl,rpath adds path for runtime libraries



Dynamic Loading

void \*handle;

double (\*cosine)(double);

char \*error;

handle = dlopen ("/lib/libm.so.6", RTLD\_LAZY);

if (!handle) {

// Handle error

// exit(1);

}

cosine = dlsym(handle, "cos");

if ((error = dlerror()) != NULL) {

// Handle error

// exit(1);

}

printf ("%f\n", (\*cosine)(2.0));

dlclose(handle);

A sample program

On success, dlopen() returns a non-NULL handle for the loaded library. On error, returns NULL.

dlsym() returns NULL for error

On success, dlclose() returns 0; on error, it returns a nonzero value

Saves error to dlerror()

Makefile Structure and Logic

A target is usually the name of a file that is generated by a program

A prerequisite is a file that is used as input to create the target

A recipe is an action that make carries out

Before make can fully process this rule, it must process the rules for the files that it depends on (the prerequisites)

The recipe will be carried out if

Any file named as prerequisites is more recent than the target file

If the target file does not exist at all

Executing Makefile

When you run `make` command, the first target in your Makefile

By default, it will look for a file called “Makefile”, use –f to override

To run a specific target instead of the first, use $ make [target name]

Example: make install

.PHONY target

By default, make thinks that the target name is a file

Imagine you have a rule `make clean` to delete files, yet you have a file named “clean” in your directory

Now `make clean` will tell you nothing to make (why?)

Use phony target to enforce `make clean`

Makefile Variable

[name] = [value]

To use it later, use $(name)

It is conventional to have variables like CC, CFLAGS, etc.

Automatic variables

$@: Target name

$<: First prerequisite

$?: All prerequisites newer than the target

$^: All prerequisites

Function Attributes

Used to declare certain things about functions in your program

Help compiler do things

Also used to control memory placement, code generation options, or call/return conventions

Take the form \_\_attribute\_\_((\_\_my\_attribute\_\_))

Initializer and Finalizer

\_\_attribute\_\_ ((\_\_constructor\_\_))

run when dlopen() is called

\_\_attribute\_\_ ((\_\_destructor\_\_))

run when dlclose() is called

Example:

\_\_attribute\_\_ ((\_\_constructor\_\_))

void to\_run\_before (void) {

printf("pre\_func\n");

}

GCC Flags:

u -fPIC: Compiler directive to output position independent code, a characteristic

required by shared libraries.

u -llibrary: Link with "liblibrary.a”

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

u -L: At compile time, find the library from this path.

u -Wl,-rpath,$path: -Wl passes options to linker.

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

u -c: Generate object code from c code but do not link

u -shared: Produce a shared object which can then be linked with other objects to

form an executable.

Commands:

1)gcc –Wall –fPIC –c ctest1.c ctest2.c

2)gcc –shared –o libctest.so ctest1.o ctest2.o

3)path=$PWD

4) gcc –Wall prog.c –lctest –o first.out (should give error)

5) gcc –Wall prog.c –lctest –L$path –o first.out

6) ./first.out (Will not execute)

7) ldd first.out

8) gcc –Wall prog.c –Wl,-rpath,$path –lctest –L$path –o first.out

9) ./first.out (Will execute)

Dynamic loading is a mechanism by which a computer program can, at run

time,

u load a library into memory

u retrieve the addresses of functions/variables contained in the library,

u execute those functions/access those variables

u unload the library from memory

u Dereference at runtime

u Not Loadtime!

u Need to use functions from <dlfcn.h>

3 mechanisms by which your program can use other software in it

u Static Linking

u Dynamic Linking

u Dynamic Loading

u Static Linking

u Dependency’s actual object code becomes part of the total executable

u Dynamic Linking

u Dependency’s object code address becomes part of the executable which is

dereferenced at load time before execution begins

u Dynamic Loading

u Dependency’s object code address becomes part of the executable which is

dereferenced at run time after execution begins

Library Naming Convention

u Libraries are prefixed with lib. When linking the library name will not contain

the library prefix

u Example: gcc src-file.c -lm –lpthread

u The libraries referenced during linking are

u the math library ("m") found in /usr/lib/libm.a

u the thread library ("pthread") found in /usr/lib/libpthread.a.

Static Libraries

u Use ar command to create a static library (ar stands for archive)

u Flags:

u -c : create the archive

u -v: verbose (shows filenames processed)

u -q: quick append to the archive (without replacement)

u -r: append to the archive with replacement

u -t: display contents of archive

**Labs 7 and 9: Git**

Software Development Process

Code is modified many many times

Adding new features

Fixing bugs

Improving performance

Many people may work on the same project

Could even be modifying same files and functions

Often many versions of software are released

Ubuntu 18.04 and 18.09 and 16.04

iOS 13 and iOS 12 and …

GCC 9.X and 5.X

Need to be able to update support old versions

GCC 6.5 released after GCC 8.2!

Source/Version Control

Helps handle the challenges of software development

Tracks changes to the files in a project

What files were added / removed?

What changes were made?

Which version had what changes?

Who made the changes

Allows you to view the full history of the project

Why Use Version Control?

Easier collaboration

Easier to go back when you mess up

Easier to work on multiple features at once

Version Control Systems (VCS)

Software that does version control

Works for more than just code! Images! Data!

Even really big things

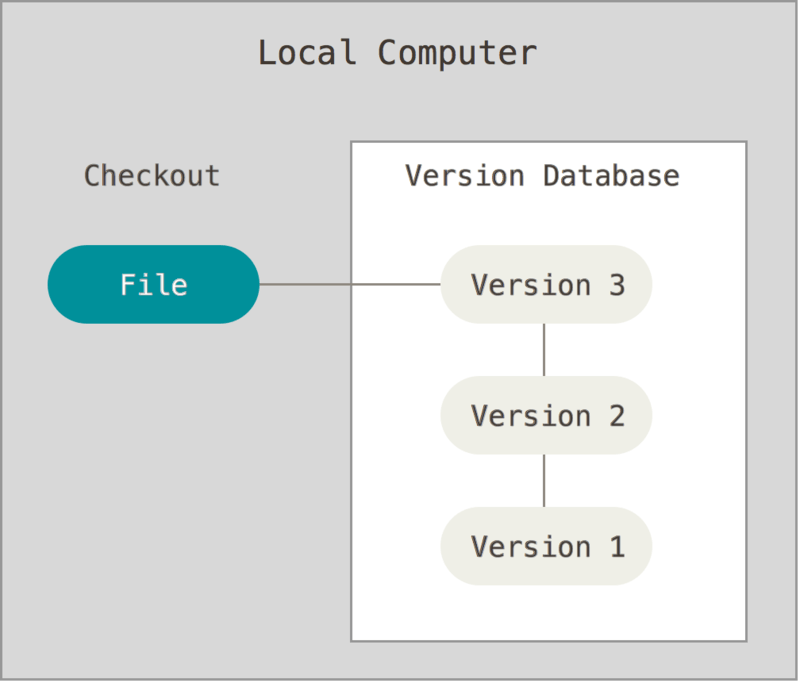
Many options available

Git, Mercurial, Subversion, Perforce

Different VCS have different design philosophies

Local VCS

At its simplest, organize different versions as folders on the local machine, Or use a database, No server needed, This is how RCS worked, Collaboration is hard, How do multiple people work with the code?, Local databases everywhere (ouch)



Centralized VCS

Addresses collaboration issues

No more local databases!

Centrally administrated

Version history kept on a central server

Users pull a copy of files to their machine to do work

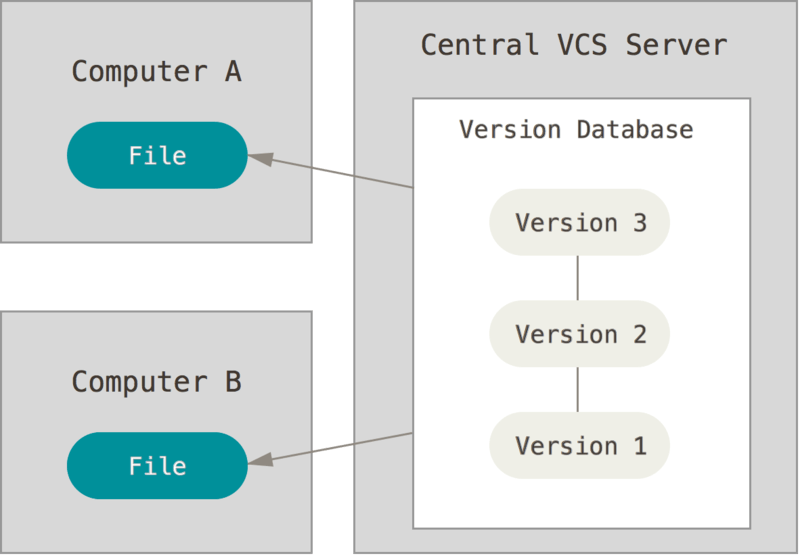
Send changes back to the server

If server has problems, you have a bad time

Lose all data :(

Used in a few systems

Subversion, Perforce, CVS



Distributed VCS

Every user’s machine has replica of version history

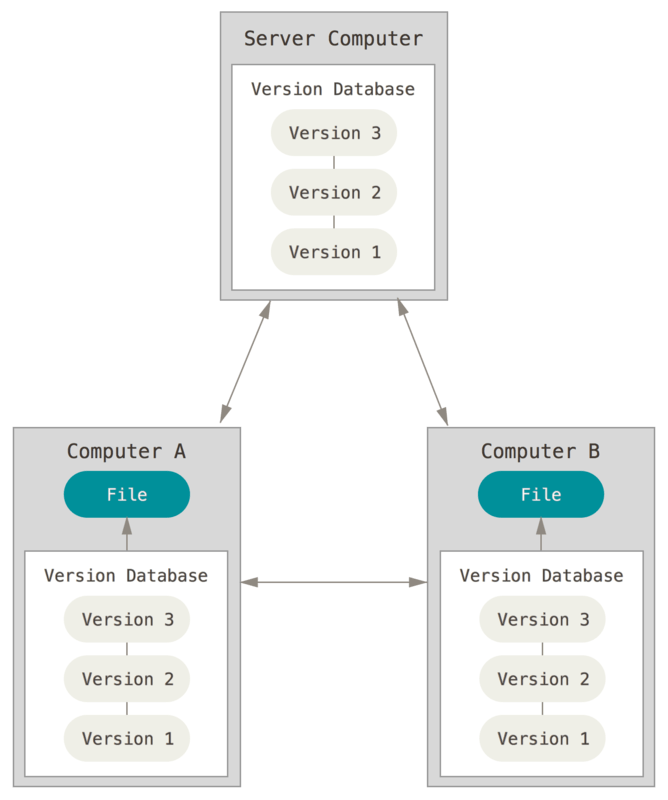
Can work even if central server is down

Central server going down doesn’t mean full loss of project

Users can communicate changes directly with one another

This is what git does!

Mercurial too



Git

Doesn’t stand for anything

Pretty popular

Git is not github

Can use Git just fine without it

Used by projects you may know

Chromium

Android

TensorFlow

More commands than you could ever know

Lots of great guides and cheat sheets

man git (mostly kidding)

Why Git?

Quick for anything you’ll likely do

Problematic at huge scales, like if you’re Microsoft

Still works for projects like the Linux Kernel though!

Design is relatively simple

Branches!

Allows non-linear development

Fully distributed

Git Basics

Check out some code

Get a copy of the files

Could be from your machine

Could be from someone else’s

Make your changes

Stage the changes you want to commit

Which changes you want to track

Commit your staged changes

Like hitting “save”

Share your changes

Git States

Files can exist in three main states

Modified

File changed but not committed

Staged

Modified and marked to be committed

Committed

Safely stored in database

Hello Git

# Make a directory for your repository

tylerdavis@tylers-macbook-pro Desktop % mkdir my\_repo

tylerdavis@tylers-macbook-pro Desktop % cd my\_repo

# Initialize an empty git repository in this directory

# This is all hidden in a .git directory

tylerdavis@tylers-macbook-pro my\_repo % git init

Initialized empty Git repository in /Users/tylerdavis/Desktop/my\_repo/.git/

tylerdavis@tylers-macbook-pro my\_repo % echo "Woah, git" > hello.txt

# Stage all modified files

tylerdavis@tylers-macbook-pro my\_repo % git add .

tylerdavis@tylers-macbook-pro my\_repo % git commit -m "Initial commit of repo"

[master (root-commit) c936e7c] Initial commit of repo

1 file changed, 1 insertion(+)

create mode 100644 hello.txt

Git Commands

Repository creation

$ git init (Create a new repository)

$ git clone (Create a copy of an existing repo)

Branching

$ git branch <new\_branch\_name>

$ git checkout <tag/commit> -b <new\_branch\_name>

Commits

$ git add (Stage modified/new/deleted files)

$ git commit (Save changes to repository)

Git Commands

Getting information

$ git status (Shows state of modified files, new files, etc.)

$ git diff (Compare different versions of files)

$ git log (Shows history of commits)

$ git show (Shows object in the repository)

Help

$ git help

More Hello Git

tylerdavis@wifi-131-179-25-45 my\_repo % echo "Git is cool" >> hello.txt

tylerdavis@wifi-131-179-25-45 my\_repo % git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: hello.txt

no changes added to commit (use "git add" and/or "git commit -a")

More Hello Git

tylerdavis@wifi-131-179-25-45 my\_repo % git diff

diff --git a/hello.txt b/hello.txt

index 4296237..ec734c6 100644

--- a/hello.txt

+++ b/hello.txt

@@ -1 +1,2 @@

Woah, git

+Git is cool

More Hello Git

# Stages the changes in hello.txt

tylerdavis@wifi-131-179-25-45 my\_repo % git add hello.txt

tylerdavis@wifi-131-179-25-45 my\_repo % git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: hello.txt

# Empty as our copy is the same as the staged one

tylerdavis@wifi-131-179-25-45 my\_repo % git diff

tylerdavis@wifi-131-179-25-45 my\_repo % git diff HEAD

diff --git a/hello.txt b/hello.txt

index 4296237..ec734c6 100644

--- a/hello.txt

+++ b/hello.txt

@@ -1 +1,2 @@

Woah, git

+Git is cool

More Hello Git

tylerdavis@wifi-131-179-25-45 my\_repo % git commit -m "Added love for git"

[master 20449e1] Added love for git

1 file changed, 1 insertion(+)

tylerdavis@wifi-131-179-25-45 my\_repo % git log

commit 20449e125b465573f17764e2a785cd112a36926a (HEAD -> master)

Author: Tyler Davis <tylerdavis@berkeley.edu>

Date: Sun Nov 3 14:59:50 2019 -0800

Added love for git

commit c936e7c373cd78a24c5201b792a1029f62cc4a31

Author: Tyler Davis <tylerdavis@berkeley.edu>

Date: Sun Nov 3 14:41:07 2019 -0800

Initial commit of repo

Git Repo Structure

Git Repository Objects

u Blobs (Binary Large Objects):

u When we git add a file such as example\_file.txt, git creates a blob object containing the

contents of the file. Blobs are therefore the git object type for storing files.

u The file's SHA-1 hash is computed and stored

u Trees

u The tree object contains one line per file or subdirectory, with each line giving file

permissions, object type, object hash and filename. Object type is usually one of “blob”

for a file or “tree” for a subdirectory

u Commit

u The commit object contains the directory tree object hash, parent commit hash, author,

committer, date and message.

u Tags

u The tag object type contains the hash of the tagged object, the type of tagged object

(usually a commit), the tag name, author, date and message

u Objects uniquely identified with hashes

Snapshot

A commit corresponds to a snapshot

Snapshot is a picture of your repo at the time you commit

If file is unchanged since last snapshot, just point at its last version

Tree

Think a “collection of files”

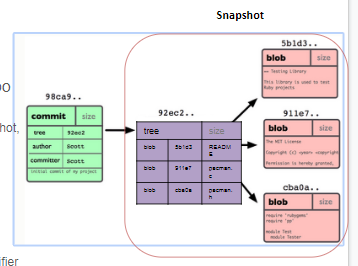
Blob

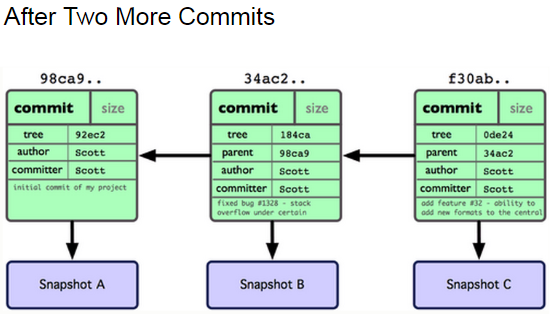
A version of a file

Checksum

Run SHA1 on object to get an identifier

This is how git refers to the object





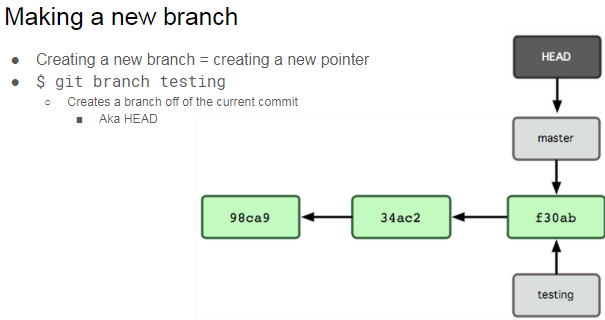
What is a branch?

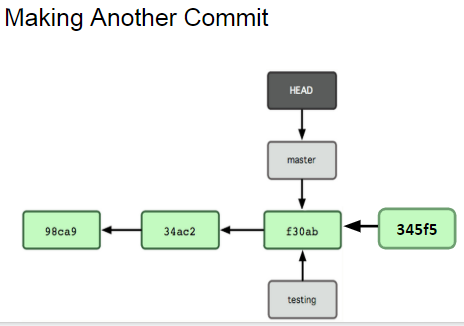
tylerdavis@wifi-131-179-25-45 my\_repo % git status

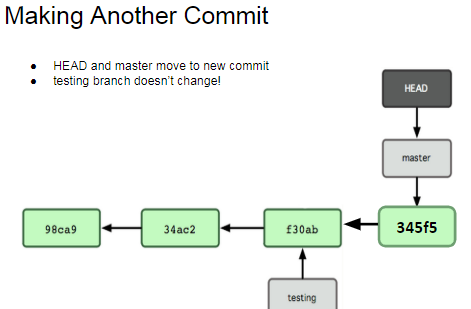
On branch master

A pointer to a commit in your repo + its history

“Master” branch automatically created when repo is created







Switching Branches

$ git checkout <branch\_name>

Why Use Branches?

Develop without affecting Master

Developing new features

Experimenting

Maintain different versions of a project

1.5 branch, 1.6 branch

Integrating Changes

Eventually you’ll want to combine changes in one branch back into Master

Two main ways to integrate changes from one branch into another

Merge

Simple, straightforward

Rebase

Much cleaner (in terms of repo history)

Git Merge

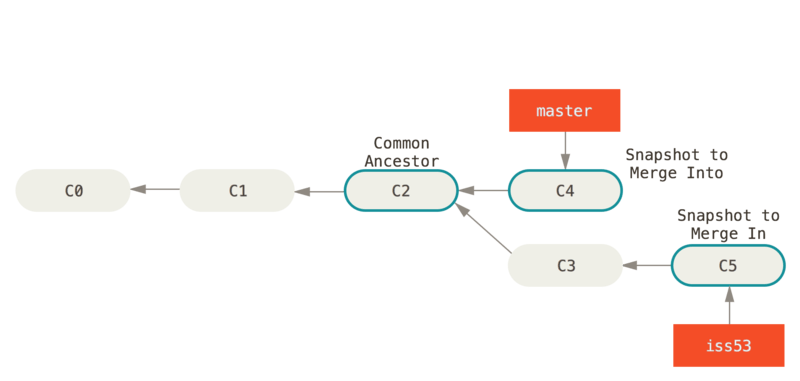
Puts a merge commit in your history

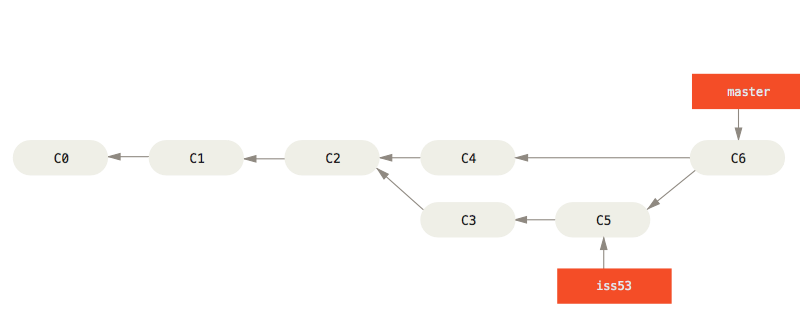
Created via 3 way merge between common ancestor and snapshots C4 and C5

Puts a merge commit in your history

Created via 3 way merge between common ancestor and snapshots C4 and C5

Con: Merge commits everywhere

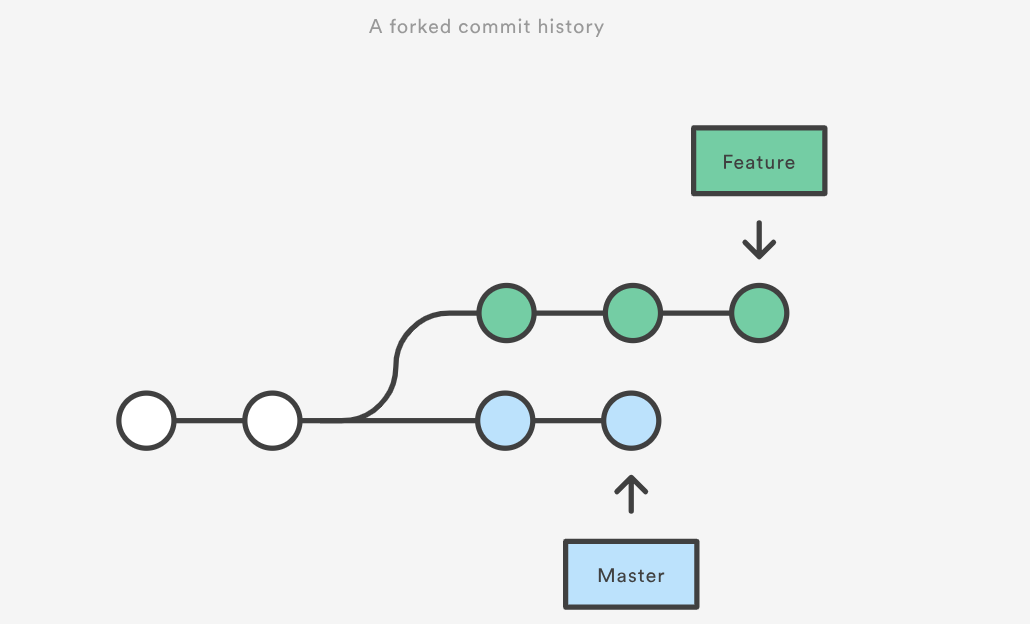


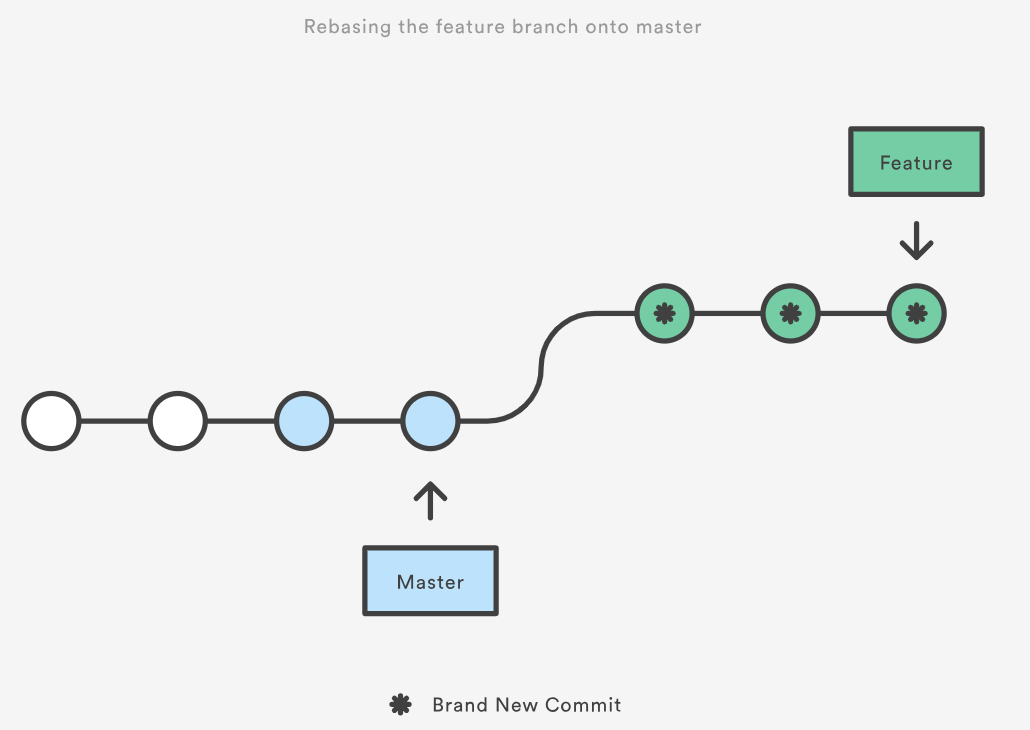


Git Rebase

Moves the “base” the branch you’re integrating with from the common ancestor to HEAD

Cleaner, but you rewrite history





Remote Repositories

You might want to interact with a repository elsewhere

On the local network, On GitHub, Good for collaboration

A bit of an offsite backup

Git clone automatically adds an “origin” repository

$ git remote

List info about remotes, Lets you manage them

Add

Remove

$ git remote show origin

Shows info about origin

Remote Branches

Branches that correspond to remote branches

Git helps you keep them in sync

Take the form <remote>/<branch>

E.g. origin/master

Can make new branches that track remote as well

$git checkout -b <branch> <remote/branch>

$git checkout --track <remote>/<branch>

Working With Remotes

$git fetch <remote>

Pull all the info about <remote> to your local

Doesn’t do any merging

<remote> is origin if not specified

$git pull <remote> <branch>

$git pull origin master

Grabs changes from the remote, and merges them into current branch

Like git fetch followed by git merge

Working With Remote

You’ve made changes locally, how do you send them to the remote?

$ git push

Sends all your changes to the remote repository

If there is a conflict, will alert you

Need to resolve conflicts locally, then reattempt push

Push might be restricted

Only certain users can push to a repo/branch

Working With Branches

tylerdavis@wifi-131-179-25-45 my\_repo % git checkout -b new\_branch

Switched to a new branch 'new\_branch'

# Show branches, and which we’re on

tylerdavis@wifi-131-179-25-45 my\_repo % git branch

master

\* new\_branch

tylerdavis@wifi-131-179-25-45 my\_repo % echo "big mood" >> hello.txt

tylerdavis@wifi-131-179-25-45 my\_repo % git add hello.txt

tylerdavis@wifi-131-179-25-45 my\_repo % git commit -m "Added my mood"

[new\_branch a536aa6] Added my mood

1 file changed, 1 insertion(+)

Working With Branches

tylerdavis@wifi-131-179-25-45 my\_repo % git checkout master

Switched to branch 'master'

# Bring the commits from new\_branch back into master

tylerdavis@wifi-131-179-25-45 my\_repo % git merge new\_branch

Updating 20449e1..a536aa6

Fast-forward

hello.txt | 1 +

1 file changed, 1 insertion(+)

tylerdavis@wifi-131-179-25-45 my\_repo % git log

commit a536aa6a40265bbb5753d3a7fa98618b4c8f5e28 (HEAD -> master, new\_branch)

Author: Tyler Davis <tylerdavis@berkeley.edu>

Date: Sun Nov 3 16:12:57 2019 -0800

Added my mood

commit 20449e125b465573f17764e2a785cd112a36926a

Author: Tyler Davis <tylerdavis@berkeley.edu>

Date: Sun Nov 3 14:59:50 2019 -0800

Added love for git

Merge Conflicts

Git normally merges two versions of a file automatically

Conflicts happen when same part of the file is changed two different ways in the two branches you’re trying to merge

Git makes you resolve the conflict manually

Open the file manually

Edit so it looks how you want

Stage + Commit

Some editors / IDEs help make this easier

More Git Commands

Reverting

$ git checkout HEAD main.cpp

Gets the HEAD revision for the working copy

$ git checkout -- main.cpp

Reverts the changes in the working copy

$ git revert

Reverts a commit with a new commit

Cleaning up untracked files

$ git clean

Tags

Human readable pointers to specific commits

$git tag -a v1.0 -m ‘Version 1.0’

Names the current HEAD commit as v1.0

More Git Commands

$git stash

Lets you “stash” your changes for later

Gives you a clean state directory, letting you switch branches

Can later reapply those changes

Git Tips

Odds are there’s a way to do what you want to do

Documentation, book, tutorials, cheatsheets, and Google are your friends

Keep your commits small!

One feature/change per commit

Makes it easier to tell which commit broke something, and what in that commit broke it

Easier for other people to understand your commit

Makes merge conflicts easier to deal with

Give your commits useful descriptions

“Fixed parsing error in read\_dataset()” vs “Fixed bugs”

Take time to clean your code before committing

Try and have your commits be snapshots of fully completed steps

Don’t just treat it like the save button in your editor

More Git tips

When working by yourself, you may play fast + loose with git

Need to be more rigorous when collaborating

The GitHub flow is one common approach

Main idea: master branch should always be ready to be pushed to production

Never put bad code in master

To implement a new feature / fix

Make a branch with a good name: “tyler\_fix\_parsing”

May need to use a fork and pull model if you don’t own the project

Implement your fix, commit along the way

Open a pull request

Get feedback on your changes

Test the results of the merge

Merge back into master

GitK

Git is great, but the command line can be a real pain

GitK helps

Visualize commit graphs

Understand repo structure

Here’s a tutorial

And some missing documentation

Other GUIs exist!

Github Desktop, Git Kraken, and Git Tower all popular

Github does some of this too

GitK

SSH into seasnet with X11 forwarding enabled

ssh -X if using terminal on macOS/Linux

“X11” option if using PuTTY

Run gitk on ~eggert/src/gnu/emacs

X11 might act up

Get xquartz for Mac

Xming for Windows

If you can’t run gitk, update your path

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

Core Takeaways

VCS exists and is important

Git is one such system

Git branch

Modify files

Git add

Git commit

Git merge

HW Hints

git clone https://github.com/tensorflow/tensorflow.git

git log

git tag

git show <hash>

Git checkout v3.0 -b my\_branch

Terms used

u HEAD

u Refers to the currently active head

u Refers to a commit object

u Branch

u Refers to a head and its entire set of ancestor commits

u Master

u Default branch

What is a Branch?

u A pointer to one of the commits in the repo (head)

u When you first create a repo, are there any branches?

u Default branch named ‘master’

u The default master branch

u points to last commit made

u moves forward automatically, every time you commit

New Branch

u Creating a new branch = creating new pointer

u $ git branch testing

u Where is new branch created?

u Off the current commit

u Where is this current commit?

u HEAD

Setting your identity on git

u Command: Git config

u Git config –global user.name <Your name>

u Git config –global user.email <Your email>

Setting up a repo

u Start from scratch

u Git init

u Clone Existing Repo

u Git clone <Link to Repo>

Working Copy of the Repo

u Adding and modifying files, tracking/staging changes - git add

u Viewing state - git status

u Ignoring files - .gitignore

u Viewing differences - git diff

u Removing staged files - git rm

u Committing changes

u git commit

u skipping the staging area (git commit -a)

u Viewing commits and history

u git log

u Patch option (-p)

Collaboration on git

u Fetching data

u Git fetch

u downloads the data to your local repository

u it doesn’t automatically merge it with any of your work

u Git pull

u automatically fetches and then merge tracked remote branch into your current branch

u Pushing Data

u Git push <remote> <branch>

Tagging

u Human readable pointers to specific commits

u Why do we need tags?

u Additional information for a commit

u Types of tags

u Annotated

u Lightweight

u Example

u git tag –a v1.0 –m ‘Version 1.0’

u This will name the HEAD commit as v1.0

u -a makes it an annotated tag

u git tag v1.4-lw

u Lightweight tag (Do not supply any flags)

More git commands

u Reverting

u git checkout HEAD main.cpp

u Gets the HEAD revision for the working copy

u git checkout – main.cpp

u Reverts changes in the working directory

u git revert

u Reverting commits (this creates new commits)

u Cleaning up untracked files

u git clean

**Lab 9 (Lab 7, Git, continued)**

Revision Selection - Short SHA-1

Commits can be referred to by their 40-character SHA-1 hash, or any unambiguous prefix (at least 4

characters long) thereof.

Revision Selection - Branch References

If a commit, say with hash abcd, is at the head of a branch, say branch b1, then commands like

git show abcd

and

git show b1

are equivalent

Revision Selection - Ancestry References

Suppose we have the following commits:

$ git log --pretty=format:'%h %s' --graph

\* 2e25043 Merge pull request #18 from ...

|\

| \* 4950521 fix sim by normalizing SNP columns

| \* 69402cd generate g effects

|/

\* c455717 tabulate\_output.py

\* f3fe695 Merge pull request #17 from ...

A caret ^ at the end of a reference refers to the parent of

that commit. e.g. c455717^ will refer to f3fe695

For merge commits such as 2e25043,

2e25043^ will refer to its first parent, c455717

while 2e25043^2 will refer to its second parent 4950521.

Moreover, since HEAD points to 2e25043,

HEAD^ and 2e25043^ are equivalent.

A tilde ~ also refers to the first parent, so HEAD^ and

HEAD~ are equivalent.

HEAD~2 refers to the first parent of the first parent of

HEAD, and the same pattern goes for HEAD~3 etc.

Revision Selection - Commit Ranges

abcd12..abcd89 selects the commits from but excluding abcd12 to abcd89 inclusive, e.g.

git log abcd12..abcd89

will show commits from abcd12 exclusive to abcd89 inclusive.

Revision Selection - Commit Ranges

git log master..experiment

will show commits in experiment not reachable from master

so the output will be

D

C

On the other hand,

git log experiment..master

will output

F

E

Rewriting History

git commit --amend can change the last commit content and commit message.

Just git add or remove the appropriate changes and call git commit --amend

Then rewrite the commit message and save/exit the editor dialog box.

Warning: Only rewrite the history when the changes do not affect commit that have been pushed to the remote.

Rewriting History

In order to rewrite the last few commits, use the git interactive rebase, e.g.

git rebase -i HEAD~3

will rebase the last 3 commits onto the 4th oldest commit.

You will see an editor open up with texts like the ones in the next slide.

Beware the the commits are listed in the reverse order to that of git log.

Three Trees

HEAD

Last commit snapshot, next parent. HEAD is the pointer to the current branch reference, which is in turn a

pointer to the last commit made on that branch.

Index

Proposed next commit snapshot. The index is your proposed next commit. We’ve also been referring to

this concept as Git’s “Staging Area” as this is what Git looks at when you run git commit.

Working Directory

The actual directories with files that you can modify with an editor.

git reset

If HEAD refers to the master branch,

and there are three commits on the master branch eb43bf8, 9e5e6a4, 38eb946,

with 38eb946 being the most recent,

then git reset 9e5e6a4 will perform the following steps.

Step 1 (Move HEAD)

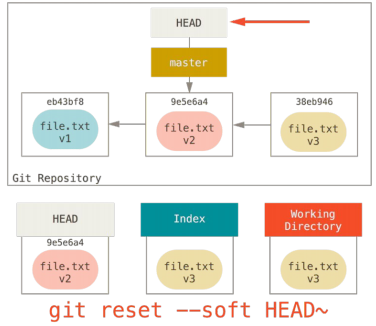
It will make the master point to the commit 9e5e6a4.

Note that if git reset --soft 9e5e6a4 is used

instead, git reset will stop at this step.

At this point, git status will show the changes file.txt

v3 as staged, since the index didn't change.



Step 2 (Updating the Index)

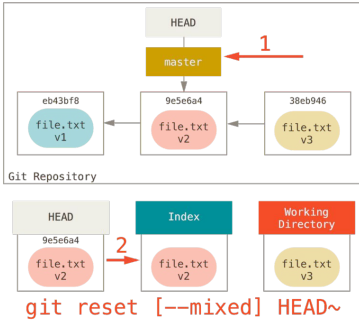
The index will be updated to be the same as the state in commit 9e5e6a4 i.e. it also unstaged everything.

However, the working directory wasn't modified, so file.txt still has the content of the version 3, i.e.

file.txt v3, and git status will show that file.txt as modified.

Note that without the --hard option, git reset will stop at this step.

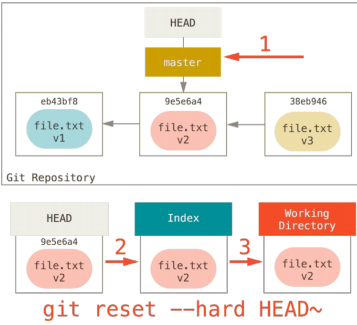
--mixed means stopping at this step, but it's the default anyways.



Step 3 Updating the Working Directory (--hard)

git reset --hard 9e5e6a4 will reset the changes to the actual files that have been modified from their states in the commit 9e5e6a4, i.e. the content of file.txt will become file.txt v2

This is a potentially dangerous operation because you will lose all the changes to file.txt at version 3.

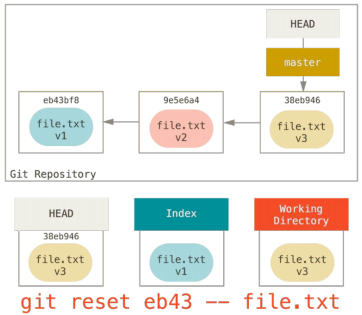


git reset at the File Level

git reset eb43bf file.txt

will not move the HEAD pointer.

You can also use the --hard option on this.



Git Plumbing Commands

Plumbing commands refer to a set of git commands that do low-level work and are not usually used for

ordinary purposes.

For example, git hash-object is a plumbing command:

echo 'test content' | git hash-object -w --stdin

--stdin indicates taking input from stdin, and -w indicates writing the content into a new file in .git/objects

Git Objects

echo 'version 1' > test.txt

git hash-object -w test.txt

# returns a hash, say 83baae61804e65cc73a7201a7252750c76066a30

echo 'version 2' > test.txt

git hash-object -w test.txt

# returns another hash, say 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

find .git/objects -type f

# .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a

# .git/objects/83/baae61804e65cc73a7201a7252750c76066a30

# ...

git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30

# prints version 1

git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

# prints version 2

git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

# prints "blob", the type of the file, due to the -t option

# blob is the leaf of the directory tree, representing a regular file.

Tree Objects

Another type of git objects is a tree object, which represents directory entries.

For example, each commit will point to a tree object, which is the top directory of the repository when the commit snapshot is taken.

Commit Objects

Each commit is also stored as a git object, which will contain information such as the tree object is points

to, the parent commit(s), author, commit message, etc.

Git References

Inside .git/refs/ there are files whose names are aliases to commit hashes.

For example, .git/refs/heads/master will contain the commit hash of the commit pointed to by the master

branch.

Another set of references are the remotes.

For example, refs/remotes/origin/master will contain the commit hash pointed to by the master branch in

the remote named origin.

Git Refspec

When we do something like git remote add origin some-url, an entry is created in the .git/config file:

[remote "origin"]

url = some-url

fetch = +refs/heads/\*:refs/remotes/origin/\*

The fetch has the format +<source>:<destination>, where <source> is the references on the remote, and

<destination> is where those references will be tracked locally.

The + sign means update the local references even if the remote branch update is not a fast forward, e.g.

if originally we have commits A <-- B (origin/master)(master), then somebody viciously changed history

on the remote to A <-- C, the + sign says this is okay and update the local references to be

A --- B <--(master)

\

C <--(origin/master)

Git Refspec

$ git log origin/master

$ git log remotes/origin/master

$ git log refs/remotes/origin/master

are all equivalent, because git expands all of them to refs/remotes/origin/master

Pushing Refspecs

Sometimes we might want to push a local branch/reference to a different reference on the remote. For

example, if we want to push the local master branch to the qa/master branch in the origin remote (qa is a

namespace, say for the QA team), we can do :

git push origin master:refs/heads/qa/master

To have a permanent mapping for fetch and push, we can update the .git/config file to

[remote "origin"]

url = some-url

fetch = +refs/heads/\*:refs/remotes/origin/\*

push = refs/heads/master:refs/heads/qa/master

Partially Ordered Set (Poset)

A binary relation "<=" on a set S is a partial order if "<=" is

reflexive, i.e. a <= a for all a in S

antisymmetric, i.e. a <= b and b <= a implies a = b

transitive, i.e. a <= b, b <= c implies a <= c

For example, for a rooted tree, if we define a <= b if a is an ancestor of b, then this "<=" is a partial order on

the nodes of the tree. Note that in general only a subset of the pairs are comparable, e.g. in the tree

b ← c ← d ← e ← f ← g

\

h ← k ← m

with b being the oldest ancestor, we have c <= f and c <= k, but neither e <= k nor k <= e is true, since

neither is an ancestor of the other.

Total Order

If a partial order can be applied to every pair of elements in the set S, then it is a total order.

That is, for every pair a, b, if either a <= b, or b <= a or both, then "<=" is a total order.

For example, the usual less-than-or-equal-to relation "<=" on the set of integers is a total order.

Topological Order

A topological order on a directed acyclic graph (DAG) G = (V, E) is a total order on all of its vertices such

that if there is a directed edge from v1 to v2, then v1 < v2.

We can view a topological ordering/sorting on G as arranging the vertices on a horizontal line such that all

edges go from left to right.

Topological Sort

One way to perform a topological sort on a DAG G is to run DFS on G, and as each vertex is finished being

processed, i.e. turned black, append it onto a FIFO queue.

The resulting queue will contain a topological ordering of the vertices, where the first out element is the

smallest and so on.

Topological Sort on a Rooted Tree

The previous algorithm for topological sorting can be simplified when the graph is a rooted tree.

Instead of insert the vertex into the FIFO queue when it turns black, we can insert a vertex onto a stack as

soon as it is discovered (turned gray).

In the end, the vertex at the top of the stack will be the smallest and so on.

Think about why this is the case.

**Lab 8 (SSH):**

Communication Over the Internet

What type of guarantees do we want?

Confidentiality, Message secrecy, (Data) Integrity, Message consistency, Authentication

Identity confirmation, Also authorization

Specifying access rights to resources

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 Encrption

Same secret key used for encryption and decryption

Example : Data Encryption Standard (DES)

Caesar's cipher

Map the alphabet to a shifted version

ABCDEFGHIJKLMNOPQRSTUVWXYZ DEFGHIJKLMNOPQRSTUVWXYZABC

Plaintext – SECRET. Ciphertext – VHFUHW Key is 3 (number of shifts of the alphabet)

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

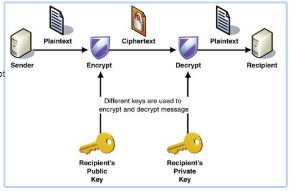
Anyone can encrypt message, but they cannot decrypt the ciphertext

Decryption

Use private key to decrypt messages

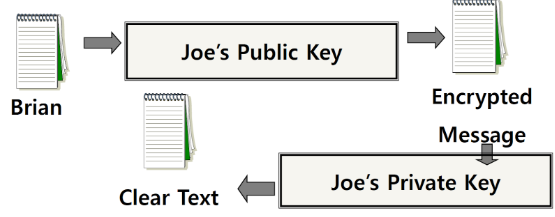
Property used - Difficulty of factoring large

integers to prime numbers



Public Key (asymmetric) Cryptography

Two keys are used: a public and a private key. If a message is encrypted with one key, it has to be decrypted with the other.



What is SSH?

Secure Shell

Used to remotely access shell

Successor of telnet

Encrypted and better authenticated session

High-Level 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 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

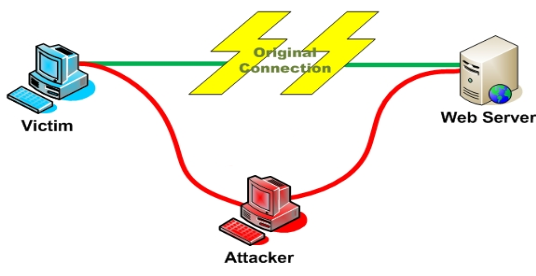
After accepting, public key is saved in ~/.ssh/known\_hosts

Host Validation

Next time client connects to server

Check host’s public key against saved public key

If they don’t match



Host Validation (cont’d)

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 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

User 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 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)

Server Steps

Generate public and private keys

$ ssh-keygen (by default saved to ~/.ssh/is\_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>

$ sudo mkdir .ssh

Change ownership and permission on .ssh directory

$ sudo chown –R username .ssh

$ sudo chmod 700 .ssh

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.

Client Steps – Make logins convenient

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

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 Addresses

$ ifconfig

configure or display the current network interface configuration information (IP address, etc.)

$ hostname –I

gives the IP address of your machine directly

$ 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

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:

Generate a Message Digest

The message digest is generated using a set of hashing algorithms

A message digest is a 'summary' of the message we are going to transmit

Even the slightest change in the message produces a different digest

Create a Digital Signature

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

Attach digital signature to message and send to receiver

Steps for Generating a Digital Signature

RECEIVER:

Recover the Message Digest

Decrypt the digital signature using the sender’s public key to obtain the message digest generated by the sender

Generate the Message Digest

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

Compare digests (the one sent by the sender as a digital signature, and the one generated by the receiver)

If they are not exactly the same => the message has been tampered with by a third party

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.

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

GNU privacy guard (> gpg [option])

u --gen key generating new keys

u --armor ASCII format

u --export exporting public key

u --import import public key

u --detach-sign creates a file with just the signature

u --verify verify signature with a public key

u --encrypt encrypt document

u --decrypt decrypt document

u --list-keys list all keys in the keyring

u --send-keys register key with a public server/-keyserver option

u --search-keys search for someone’s key

