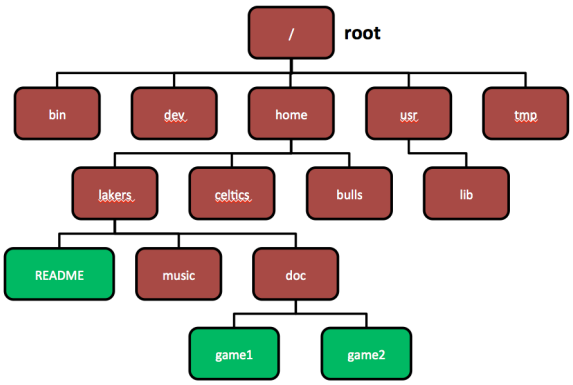
Week 1: Introduction, files, and editing

Command line	GUI
Steep learning curve	Intuitive
Pure control	Limited control
Cumbersome multitasking	Easy multitasking
Convenient remote access	Bulky remote access

- GNU/Linux
  - kernel
    - core of operating system
    - allocates time and memory to programs
    - handles file system and communication btwn software and hardware
  - shell
    - interface btwn user and kernel
    - accepts commands as text, interprets them
    - causes commands to be carried out using OS API
    - common shells: bash, csh, kWh
  - programs
- Files and processes
  - process
    - an executing program
    - identified by PID
  - file
    - collection of data
    - document, executable, directory, device, etc.



/bin: binaries, executables
/home: user home directories
/sbin: system binaries (tools used by admin)
/tmp: temporary files, removed routinely
/dev: devices (urandom, null, stdin, stdout)

- redirection
  - > file: write stdout to a file

```
ls > out.txt
   sends ls to out.txt
ly > out.txt
   prints error message on screen, out.txt empty
ly > out.txt 2>&1
    // send error message (2) to out.txt as well
```

  - >> file: append stdout to a file
  - < file: use contents of a file as stdin
  - 1: stdout, 0: stdin, 2: stderr

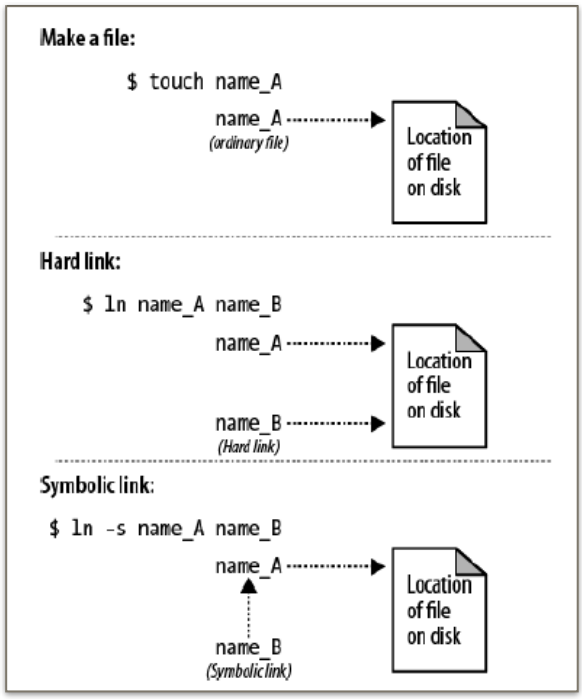
Linux file permissions: chmod	
rwx rwx rwx u g o	read write executable user group others
Reference	
u (user)	owner of the file
g (group)	users who are members of the file's group
o (others)	users who are neither the owner nor members of the group
a (all)	all three of the above (ugo)
Mode	
r (read)	read a file or list directory's contents
w (write)	write to a file or directory
x (execute)	execute a file or recurse a directory tree
Operators	
+	add the specified modes to the specified classes
-	remove the specified modes from the specified classes
=	the modes specified are to be made the exact modes for the specified classes
Numeric chmod	
7	full
6	read and write
5	read and execute
4	read only
3	write and execute
2	write only
1	execute only
0	none

usage:
chmod [references] [operator][modes] file1
	`chmod ug+rw mydir // symbolic`
	`chmod a-w myfile`
	`chmod ug=rx mydir`
	`chmod 664 myfile // numeric`

- pipes
  - take output of one and use it as input to the other
  - without pipes, use a bunch of temporary files

```
cat page.html | tr -c 'A-Za-z' '[\n*]' | sort -u | comm -23 -words
```

- create a link using ln
  - hard links: point to physical data
  - soft links (symbolic links -s): point to a file



- touch
  - update access and modification time to current time

```
touch filename
touch -t 201101311759.30 // changes filename's access & modification time to 2011 January 31 17:59:30
```

- man
  - read a man page for a Linux command

```
man <command_name>
man section command_name
```

  - headings
    - name of command (ls – list files)
    - synopsis (cp [options] ... source dest)
    - description (shows options and their descriptions)
    - other headings: authors, bugs, examples, copyright

- daemon
  - process that runs in the background
  - example: cron can run a full backup every month

Command	Description
cd	change working directory
~	home directory
/	root directory
ls	list contents of a directory
-d	list only directories
-a	all files including hidden
-l	long listing including perm info
-s	show size of each file in blocks
find	lets you search for files recursively

`find . -name "*.o"`
`find . -type d`
`find . - atime +30`
`find . -user lsamy`
`find . -perm 500`
	`all .o files in cur dir`
	`all directories`
	`all that haven't been accessed in 30 days`
	`files from certain user`
	`files with that perm`

-perm	permission of a file
-name	name of a file ? matches any single character * matches one or more characters [] matches any one of the characters between the brackets
-type	f (regular file) d (directory) l (links)
-maxdepth	how many levels to search
whereis <command>	locate the binary, source, and manual page files for a command
whatis <command>	returns name section of man page
ps	list processes currently running
kill <PID>	terminate a certain process
cat	output to stdout

`cat file1`
`cat > file2`
`cat file3 | command`
`cat file4 | grep something`	

- (as file name)	read from stdin
echo [options] [strings]	output text
`x=10` `echo value of x is $x  // outputs "value of x is 10"`	
sort [options] [file]	sort order depends on locale (C locale: ASCII)
-u	unique (only first of an equal run)
comm [op] FILE1 FILE2	compare two sorted files line by line, output 3 column 1 = unique to file1, 2 = unique to file2, 3 = common
-23	print only columns 2 and 3
tr [op] set1 [set2]	translate or delete characters
-c	complement the set of characters in set1
-s	squeeze multiple occurrences of the characters listed in the last operand into a single instance of the char

Week 2: Commands and basic scripting

- grep: search for text. Matches LINES
  - grep
    - basic regular expressions
    - meta-characters ? + { | (and) lose special meaning
    - need to escape some symbols like pipe and parentheses

      ```
      cat script.sh | egrep ^#\(TODO\|FIXME\)
      ```
  - egrep
    - extended regular expressions

      ```
      cat script.sh | egrep ^#(TODO|FIXME)
      ```
  - fgrep
    - can only match fixed strings, no regular expressions

      ```
      cat script.sh | fgrep "#TODO"
      cat script.sh | fgrep "#FIXME"
      ```

- sed: substitute text
  - sed 's/regExpr/replTex/[g]'
  - g means global replacement, not just first instance
  - example

    ```
    echo "Sunday" | sed 's/day/night' // Sunnight

    cat script | sed 's/^#.*//g'
    // match any comment that starts at beginning of line, replace with
    nothing (deletes comments!)
    ```
  - save patterns using back referencing: put expression you want to save in parentheses, escape with backslashes, then do \1 (or other number indicating position if you have multiple).
  - Following example removes parentheses of all words contained in parentheses

    ```
    cat file.txt | sed 's/(\(.*\))/\1/g'
    ```
  - Following example: 1234 -> 1,234

    ```
    cat file.txt | sed 's/\([0-9]\)\([0-9]{3}\)/\1,\2/g'
    ```
  - Following example removes everything after and including the first colon

    ```
    echo $PATH | sed 's/:.*//'
    ```

- locale
  - set of parameters that define a user's cultural preferences (language, country, other area-specific things)
  - locale command prints info about current locale environment to stdout
  - gets its data from the LC_* environment variables
    - LC_TIME: data and time formats
    - LC_COLLATE: order for comparing and sorting
      - LC_COLLATE='C' sorting in ASCII order
      - LC_COLLATE='en_US' sorting case insensitive
  - C locale is default
- environment variables
  - can be accessed from any child process
  - HOME: path to user's home directory
  - PATH: list of directories to search in for command to execute

---

- bash scripting
  - shell script file is just a file with shell commands
  - when shell script is executed, a new child "shell" process is spawned to run it
  - first line (shebang) states which child "shell" to use

    ```
    #! /bin/sh
    #! /bin/bash
    ```
  - echo: writes arguments to stdout, can't output escape characters without -e

    ```
    $ echo "Hello\nworld"     // outputs Hello\nworld
    $ echo -e "Hello\nworld"  // outputs Hello (new line) world
    ```
  - printf: can output data with complex formatting, just like C printf()

    ```
    $ printf "%.3e\n" 46553132.14562253     // outputs 4.655e+07
    ```
  - variables declared using =
  - NO TYPES AND NO SPACES ON SIDES OF EQUALS SIGN

    ```
    $ var="hello" // NO SPACES
    $ echo $var
    ```
  - return value of previous command: $?

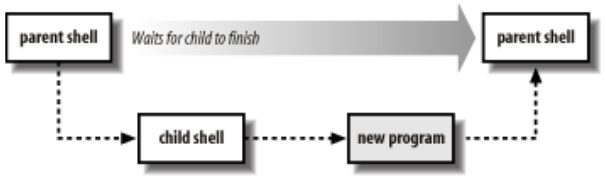
    ```
    diff file1 file2
    ```

Regular Expressions		
search for text with a particular pattern		
Quantification		
*	both	match 0 or more
+	ERE	match 1 or more
?	ERE	match 0 or 1
{n}	ERE (add \ for BRE)	Exactly n occurrences of preceding regular expression
{n,m}		Between n and m occurrences
{n,}		At least n occurrences
Grouping		
ab*	both	a, ab, abb
(ab)*	ERE	(empty string), ab, abab
Anchors		
^	both	beginning of the line
$	both	end of the line
Alternation		
. (period)	both	match any single char except null
[ab\cd]ba	both	any one of the enclosed characters (hyphen for range) aba, bba, \ba, cba, dba
(hello | hi) world	ERE	choices between words instead of characters (match reg expo specified before or after)
[^aeiou]		matches NOT vowels

POSIX	Matching
[:alnum:]	alphanumeric
[:alpha:]	alphabetic
[:blank:]	space and tab
[:digit:]	numbers
[:lower:]	lowercase
[:space:]	whitespace

RegExp	Meaning
tolstoy	"tolstoy" anywhere on a line
^tolstoy	"tolstoy" at beginning of line
tolstoy$	"tolstoy" at end of line
^tolstoy$	line containing "tolstoy" and nothing else
[Tt]	Tolstoy or tolstoy
tol.toy	tolstoy, tolttoy, tolmtoy, etc
tol.*toy	toltoy, tolstoy, tolWHOtoy, etc.

Compiled languages	Interpreted languages
Programs translated from original source code into machine code that's executed by hardware	Interpreter program (the shell) reads commands, carries out actions commanded as it goes
Efficient and fast	Much slower execution
Require recompiling	Portable
Work at low level, dealing with bytes, integers, floating point, etc.	High-level, easier to learn
Example: C/C++	Example: PHP, Ruby, bash



```
if [ $? -eq 0 ]
then
    echo “files were the same”
else
    echo “files were different”
```

- access total number of arguments: $#
- DIFFERENT FROM C AND PYTHON, WHERE TOTAL NUMBER OF ARGUMENTS INCLUDES NAME OF PROGRAM. IN BASH IT DOESN'T COUNT THE NAME!

```
if [$# -ne 2 ]
    then echo “error”
```

- accessing arguments: positional parameters represent a shell script's command-line arguments
  - for historical reasons, enclose number in braces if > 9

```
#! /bin/sh
#test script
echo first arg is $1
#echo tenth arg is ${10}
./test hello // first arg is hello
```

- if statements: use test command or []
- MUST BE A SPACE BETWEEN SQUARE BRACKETS AND INFO!
- can use -gt, -lt, -eq, etc.

```
#! /bin/sh
if [ 5 -gt 1 ]
then
    echo “5 greater than 1”
else
    echo “not possible”
fi
```

- loops
  - while loop

```
#! /bin/sh
COUNT=6
while [ $COUNT -gt 0 ]
do
    echo “Value of count is: $COUNT”
    let COUNT=COUNT-1
done
```

  - for loop: based on IFS (space by default) as delimiter
  - to change IFS, do IFS=‘,’ for example

```
#! /bin/sh
temp=`ls`
for f in $temp  // f refers to each word in ls output
do
    echo $f
done
```

Exit return	meaning
0	exited successfully
>0	failure to execute command
1-125	command exited unsuccessfully
126	command found, but file not executable
127	command not found

- quotes
  - backticks `` expand as shell commands

```
# temp=`ls`
# echo $temp // outputs result of ls command
```

  - single quotes ‘ ‘ do not expand at all, literal meaning

```
# temp=‘ls’
# echo $temp // outputs ls

# temp=‘$hello$hello’
# echo $temp // outputs $hello$hello
```

  - double quotes “” are almost like single quotes but expand backticks , \, and $

```
# temp=‘ls’
# echo $temp // outputs result of ls command

# temp=‘$hello$hello’
# echo $temp // outputs what’s stored in hello variable twice
```

Week 3: Scripting, VMs, and construction tools

- untar: tar -xzvf filename.tar.gz (-x extract, -z gzip, -v verbose, -f file)

- compiling

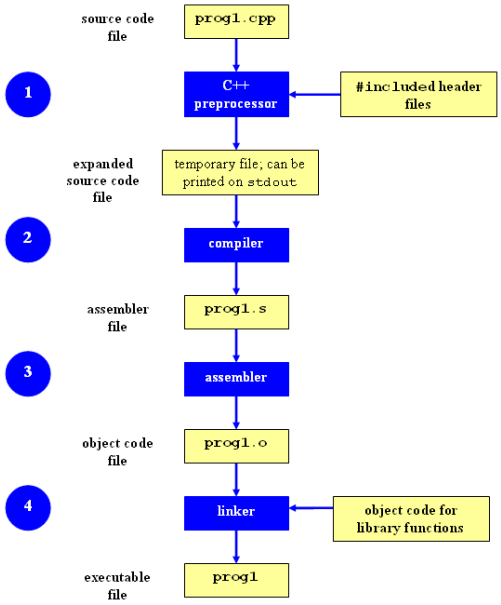
```
shop.cpp #includes shoppingList.h and item.h
shoppingList.cpp #includes shoppingList.h
item.cpp #includes item.h
gcc -Wall shoppingList.cpp item.cpp shop.cpp -o shop
```

- make only small changes -> not efficient to recompile, so avoid waste by producing a separate object file for each source file

```
gcc -Wall -c shoppingList.cpp item.cpp shop.cpp
gcc shoppingList.cpp item.cpp shop.cpp -o shop
```

- what if we change item.h? Need to recompile every source file that includes it and every source file that includes a header that includes it (item.cpp and shop.cpp)
- difficult to keep track of files when project is large, so use Make!

- configure
  - script that checks details about the machine before installation
  - creates Makefile
- make
  - utility for managing large software projects
  - compiles files and keeps them up to date
  - efficient compilation (only files that need to be recompiled)
  - requires Makefile to run
  - compiles all the program code and creates executables in current temporary directory
- make install
  - searches for a label named install within the Makefile, and executes only that section of it
  - moves the executable to system directories like /bin



- Python

```
#!/usr/local/cs/bin/python
```

  - compiled and interpreted (compiled to bytecode which is interpreted by the Python interpreter)
  - no braces or key words for code blocks, so indents are everything!
- output

```
import sys.stdout.write("Output")
 sys.write
```

- optparse library: tool for parsing command-line arguments
  - argument
    - string entered on the command line and passed into the script
    - elements of sys.argv[1:] (sys.argv[0] is name of program)
  - option: supplies extra info to customize execution
  - option argument: follows option, closely associated with it
- List
  - like a C array but dynamic (expands as new items are added) and heterogeneous (can hold objects of different types)
  - access elements: List_name[index]

```
t = [123, 3.0, 'hello!']
print t[0]  //123
print t[1]  //3.0
print t[2]  //hello!
s = [1, 2, 3]
merged_list = t + s
list.append(10)
print merged_list //123, 3.0, hello, 1, 2, 3, 10
```

- for loops: list = ['Mary', 'had', 'a', 'little', 'lamb']
- for i in range(0,10,2): is same as for (int i=0;i<10;i+=2)

for i in list: print i	for i in range(len(list)): print i
Mary	0
had	1
a	2
little	3
lamb	4

```
# 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 item.h 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
```

Rule

Comments
Targets
Prerequisites
Commands
Dependency Line

```
--- 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

```
#!/usr/bin/python

import random, sys
from optparse import OptionParser

class randline:
   def __init__(self, filename):
       f = open (filename, 'r')
       self.lines = f.readlines()
       f.close ()

   def chooseline(self):
       return
random.choice(self.lines)


def main():
   version_msg = "%prog 2.0"
   usage_msg = """%prog [OPTION]...
FILE Output randomly selected lines
from FILE."""
```

Tells the shell which interpreter to use

Import statements, similar to include statements
Import OptionParser class from optparse module

The beginning of the class statement: randline
The constructor
Creates a file handle
Reads the file into a list of strings called lines
Close the file

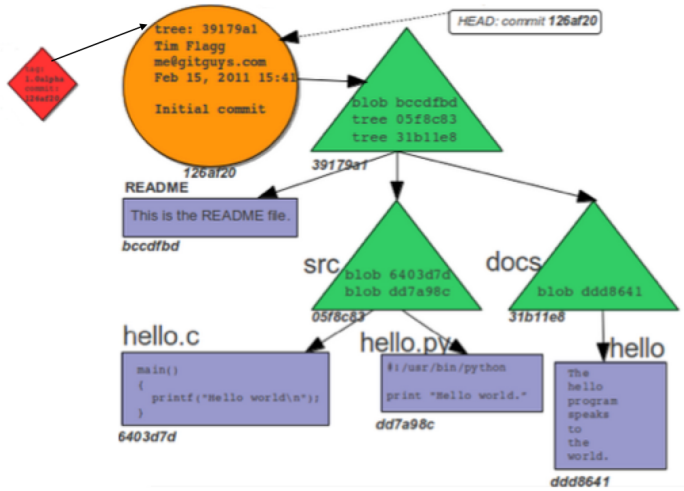
The beginning of a function belonging to randline
Randomly select a number between 0 and the size of lines and returns the line corresponding to the randomly selected number
The beginning of main function
version message
usage message

Week 4: Change management

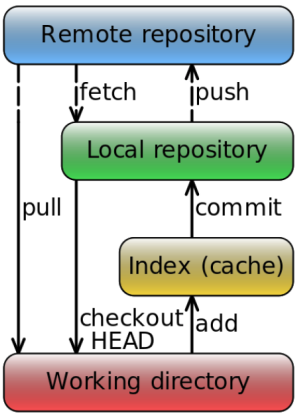
- Disadvantages of diff and patch
  - diff requires keeping a copy of old file
  - work with only 2 versions of a file
  - need to save a full copy even if only a single line changes
  - changes to one file might affect other files, so need to make sure those versions are stored together as a group

Centralized VCS (SVN, CVS)	Distributed VCS (Git)
Single central copy of the project history on a server	Each developer gets fill history of a project on their own hard drive
Changes uploaded to server, Other programmers can get changes from the server	Developers can communicate changes between each other without going through a central server
Pros: Everyone can see changes at same time, simple to design	Pros: can commit changes while offline, commands run fast, share changes with a few before showing everyone, hard to lose data
Cons: Single point of failure (no backups)	Cons: long time to download, a lot of disk space to store all versions

- version control tools - GIT!
  - track and control changes to files
  - what changes were made, when, by who, revert
  - repository
    - database usually stored on a server
    - contains set of files and directories, as well as the full history and different versions of a project
  - working copy: local copy of files from a repository at a specific time or revision
  - check-out: create a local working copy from the repo
  - commit: write the changes made in the working copy back to the repo



- git architecture: objects uniquely identified with HASHES
  - blobs
    - sequence of bytes, like files
    - stored in .git/objects
  - trees
    - like directories
    - can include other git trees or bobs
  - commits
    - created when “git commit” is executed
    - points to the top-level tree of the project at the point of commit
    - contains name of committer, time of commit, and hash of current tree
  - tags
    - give names to commit objects for convenience
    - include tag name, commit referred to, tag message, tagger info
  - head: reference to a commit object (a repo can contain any # of heads)
  - HEAD: the currently checked out head
  - branch
    - pointer to one of the commits in the repo and all ancestor commits
    - default branch is “master”
      - points to last commit made
      - moves forward automatically
    - new branches created at current commit
    - why branch?
      - experiment with code without affecting main branch
      - 2 versions of a project



- git commands

```
git init   // create new repo
git clone  // get copy of existing repo
git add    // add files to the index
git commit // changes are added to the repo
git checkout // checkout a specific version/branch of tree
git revert   // doesn't delete commit object, just applies a patch (reverts can be reverted)
getting information: git help, git status, git log, git show, git diff
```

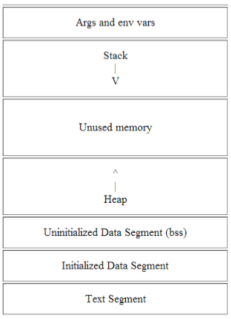
- first git repo

```
mkdir gitroot
cd gitroot
git init  // creates empty git repo
echo “Hello World” > hello.txt
git add . // must be run prior to a commit
git commit -m ‘checkin number one’
echo “I love Git” >> hello.txt
git status // shows hello.txt in list of modified files
git diff // shows changes we made compared to index
git add hello.txt
git diff // no changes now because diff compares to index
git diff HEAD // now we can see changes in working version
```

Week 5: Low-level construction and debugging

- debugger
  - program used to run and debug other programs
  - step through source code line by line
  - interact with and inspect program at run-time
  - if program crashes, debugger outputs where/why it crashed
- GDB
  - GNU debugger for C, C++, Java, Objective C
  - compiling:
    - `normally: gcc [flags] <source files> -o <output file>`
    - `debugging: gcc [flags] -g <source files> -o <output file>`

- stack info
  - every time a function is called, an area of memory is set aside for it called a stack frame. It contains
    - storage space for all local variables
    - memory address to return to when the called function returns
    - arguments/parameters of the called function
  - collectively, all stack frames make up the call stack



- C programming
  - basic data types
    - int: usually 4 bytes
    - float: usually 4 bytes
    - double: higher-precision floating point, usually 8 bytes
    - char: 1 byte
    - void
    - NO BOOL before C99
  - pointers
    - variables that store memory addresses

```
int *ptr; // ptr is pointer to int
int var = 77;
ptr = &var; // ptr points to var
*ptr = 70; // change value of var to 70
**pointerToPointer = &ptr; // pointer to a pointer
```

  - pointers to functions (functors)
    - 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: following two do same thing

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

  - structs
    - NO CLASSES IN C
    - used to package data
    - cannot have member functions
    - no such things as access specifiers in C (C++ class members have access specifiers and are private by default)
    - don't have constructors defined for them (C++ classes must have at least a default constructor)

gdb commands	
r/run	run program
breakpoints	
breakpoint/break/br/b	set breakpoint
break file1.c:6	pauses when it reaches line 6 of file1.c
break my_function	pauses when it reaches first line of my_function every time
break [position] if expression	pauses at position only when expression is true
info breakpoints/break/br/b	shows list of all breakpoints
deleting, disabling, and ignoring BP's (if no arguments provided, all breakpoints are affected!)	
delete [bp number | range]	deletes breakpoint
disable [bp number | range]	temporarily deactivates breakpoint
enable [bp number | range]	restores disabled breakpoints
ignore bp_number iterations	pass over a breakpoint without stopping a certain number of times
displaying data	
print [/format] expression	prints value of the expression in the specified format
d	decimal
x	hexadecimal
o	octal
t	binary
resuming execution after a break	
c/continue	continue executing until next breakpoint
n/next	execute next line as single instruction
s/step	same as next but step into functions
f/finish	resume execution until the current function returns
watchpoints: watch/observe changes to variables	
watch my_var	debugger will stop when my_var's value changes
rwatch expression	stop program whenever reads value of any object involved in the evaluation of expression
analyzing the stack	
backtrace/bt	shows the call stack (each frame listing gives arguments to the function and line that's being executed in that frame)
info frame	info about current stack frame including return address and saved register values
info locals	lists local variables of the function corresponding to the stack frame with their current values
info args	lists argument values of the corresponding function call

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

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

- dynamic memory
  - memory that is allocated at runtime
  - allocated on the heap

    ```
    void *malloc (size_t size);
    ```

    - allocates size bytes and returns a pointer to the allocated memory

    ```
    void *realloc (void *ptr, size_t size);
    ```

    - changes the size of the memory block pointed to by ptr to size bytes

    ```
    void free (void *ptr);
    ```

    - frees the block of memory pointed to by ptr
- reading/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 a pointer to a file, stdin, stdout, or stderr
  - format string

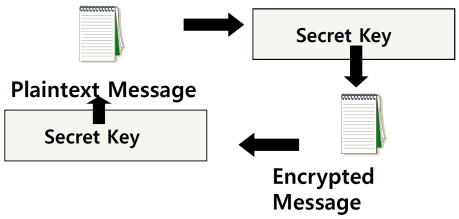
    ```
    int score = 120;
    char player[] = “Mary”;
    printf(“%s has &d points.\n”, player, score);
    ```

- compiling

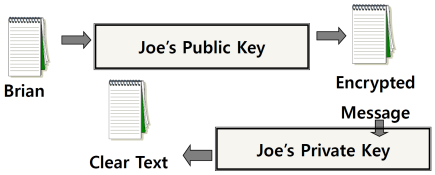
```
gcc -o FooBarBinary -g foobar.c
```

  - gcc = compiler
  - -o: indicates name of binary/program to be generated
  - -g: enable debugging using GDB
  - foobar.c is source code to be compiled

Week 6: Systems programming

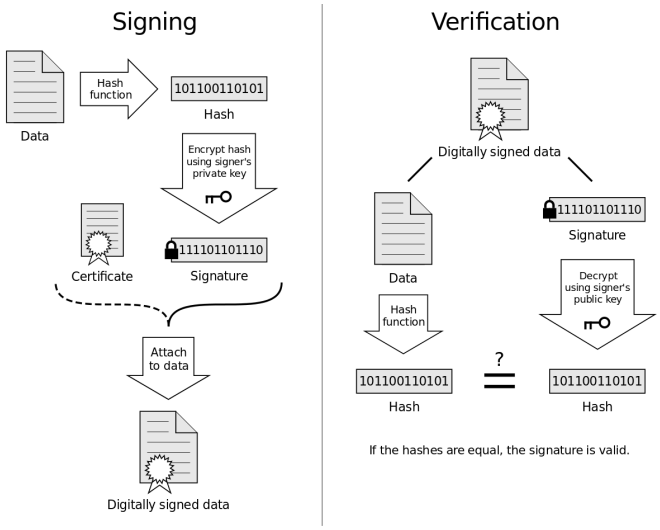


- When communicating over the Internet, we want
  - confidentiality (message secrecy)
  - data integrity (message consistency)
  - authentication (identity confirmation)
  - authorization (specifying access rights to resources)
- Encryption Types
  - Symmetric Key Encryption: key used to encrypt is the same as key used to decrypt
  - Asymmetric Key Encryption: public and private key

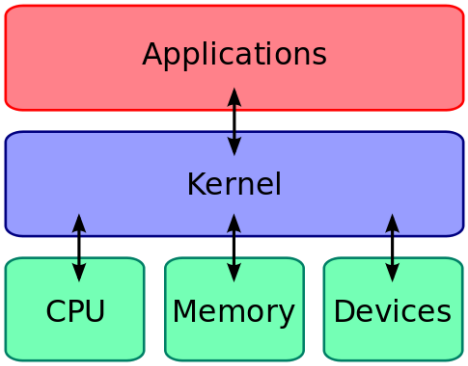


- SSH: ssh username@host
  - host validation
    - checks hosts' public key against saved public key
    - encrypt message with public key, server can decrypt it with its private key if it's the true owner
  - session encryption
    - client and server agree on a symmetric encryption key
    - all messages encrypted at sender with session key and decrypted at the receiver with session key
  - client authentication
    - password-based authentication: prompt user for password on remote server
    - key-based authentication
      - generate key pair on your computer
      - copy your public key to the server
      - server authenticates you if you have the private key
      - you can protect the private key with a passphrase, but must type your password every time
- ssh-agent
  - passphrase-less ssh
  - provides a secure way of storing the private key
  - ssh-add prompts user for passphrase once and adds it to the list maintained by ssh-agent
  - OpenSSH will talk to the local ssh-agent daemon and retrieve the private key from it automatically

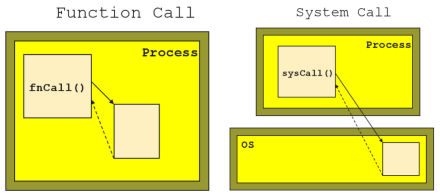
- data integrity
  - digital signature
    - electronic stamp or seal appended to document to show it hasn't changed during transmission
    - can be attached to message or detached
    - detached signature is stored and transmitted separately from the message it signs
  - encrypt hash with private key and decrypt with public key (doesn't make info confidential, but makes sure it hasn't changed)
- steps for generating a digital signature
  - sender
    - generate message digest using hashing algorithms. Even the slighted change in the message produces a different digest
    - create a digital signature by encrypting the message digest with the sender's private key
  - receiver
    - decrypt digital signature using sender's public key
    - generate the message digest using same message digest algorithm
    - compare digests. If they aren't exactly the same, the message has been tampered with
    - we can be sure that the digital signature was sent by the sender 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



Week 7: System call programming and debugging

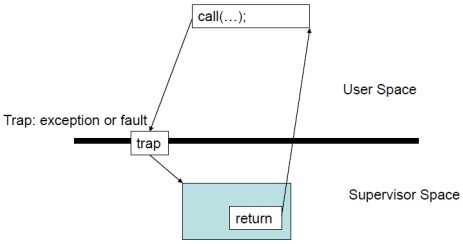


- Processor modes: place restrictions on the types of operations that can be performed by running processes
  - user mode: restricted access to resources (memory, I/O devices, CPU, etc) – must use system calls
  - kernel/supervisor mode: unrestricted access
  - why? ensure protection (don't want malicious program to cause damage to other processes) and fairness (make sure processes have a fair use of devices)
- Kernel code: only code that is trusted
  - core of the OS
  - interface between hardware and software
  - controls access to system resources
- System calls
  - part of the kernel
  - ONLY WAY a user program can perform privileged instructions
  - changes CPU's mode from user mode to kernel mode to enable more capabilities
  - when made, program is interrupted and control is passed to the kernel
- system calls are expensive and can hurt performance
  - process interrupted and computer saves its state
  - OS takes control of CPU and verifies validity of operation
  - OS performs requested action
  - OS restores saved context, switches to user mode
  - OS gives control of the CPU back to user process
- library functions
  - make system calls in more efficient matter
  - getchar vs. read
  - putchar vs. write
  - fopen vs. open
  - fclose vs. close
- buffered IO
  - unbuffered IO: every byte is read/written by the kernel through a system call
  - buffered IO: collect as many bytes as possible in a buffer and read more than a single byte into a buffer
  - decreases number of read/write system calls and the corresponding overhead
- buffered IO example
  - writing to a file
    - fwrite() copies outgoing data to a local buffer as long as it's not full and returns to the caller immediately
    - when buffer space is running out, calls the write() system call to flush the buffer to make room
  - reading from a file
    - fread() copies requested data from the local buffer to user's process as long as the buffer contains enough data
    - when the buffer is empty, fread() calls the read() system call to fill the buffer with data and then copies data from the buffer



- Caller and callee are in the same process
- Same "domain of trust"

- Control is transferred from the untrusted user process to the trusted OS

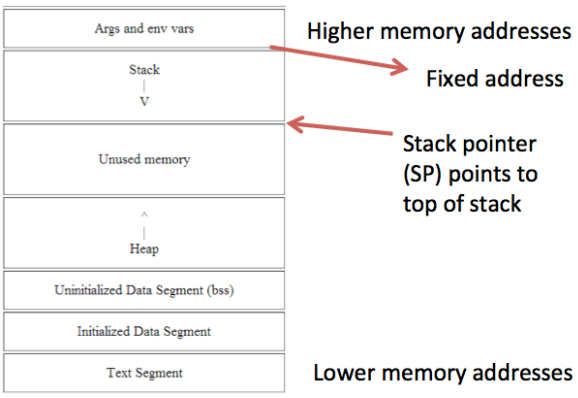


Week 8: Buffer Overflow

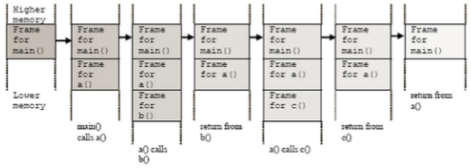
- stack
  - contains parameters to functions, local variables, and data necessary to recover program state
  - frame pointer points to find location within the frame. Variables are referenced by offsets to the FP
- constructing a stack frame
  - push the arguments
  - push return address
  - save old stack pointer
  - advance stack pointer to reserve space for local variables and state info
- buffer overflow
  - stuffing more data into a buffer than it can handle
  - can change the return address of a function to execute arbitrary code
  - place code you want to execute in the buffer, then make the return address point back into the buffer
- canary
  - random size block inserted after local variables, first thing to get corrupted
  - safeguard against buffer overflow

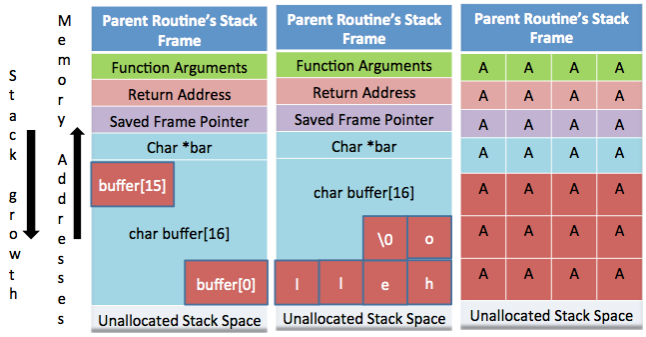
Valgrind

- tool for memory debugging
- memory leaks, use of uninitialized memory, heap/stack buffer overrun, profiling




```c
#include <stdio.h>

int a();
int b();
int c();

int a()
{
  b();
  c();
  return 0;
}

int b()
{ return 0;

int c()
{ return 0; }

int main()
{
  a();
  return 0;
}
```



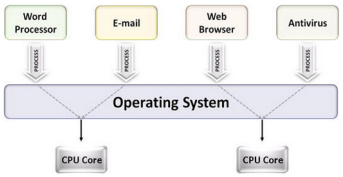


Week 9: Multithreaded Performance

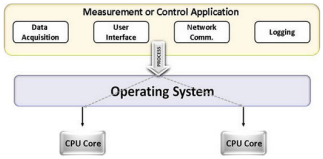
- Multiprocessing: the use of multiple CPUs/cores to run multiple tasks simultaneously
- Parallelism: executing several computations simultaneously to gain performance
  - multitasking
    - several processes schedules alternately or possible simultaneously on a multiprocessing system
    - processes insulated from each other
    - expensive creation/destruction
    - an error in one process cannot bring down another process
  - multithreading
    - same job is broken logically into pieces (threads) which may be executed simultaneously on a multiprocessing system
    - threads share the same address space
    - lightweight creation/destruction
    - easy inter-thread communication
    - an error in one thread can bring down all threads in process
- thread
  - flow of instructions, path of execution within a process
  - the smallest unit of processing scheduled by OS
  - a process consists of at least one thread
  - multiple threads can be run on
    - uniprocessor (time-sharing): processor switches between different threads (parallelism is an illusion)
    - multiprocessor: multiple processors or cores run the threads at the same time (true parallelism)
- multithreading
  - threads share all of the process's memory except for their stacks
  - data sharing requires no extra work (no system calls, pipes, etc.)
  - advantages of shared memory
    - powerful: can easily access data and share it among threads
    - more efficient: thread creation and destruction less expensive than process creation and destruction
  - WATCH OUT FOR RACE CONDITIONS!

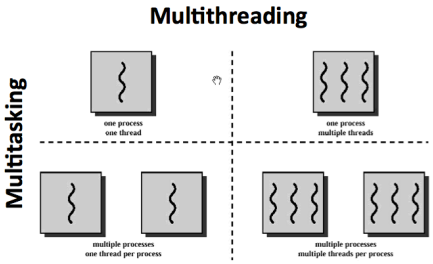
- pthread functions
  - pthread_create: create new thread within a process
  - pthread_join: waits for another thread to terminate
  - pthread_equal: compares thread IDs to see if they refer to the same thread
  - pthread_self: returns the ID of the calling thread
  - pthread_exit: terminates the currently running thread

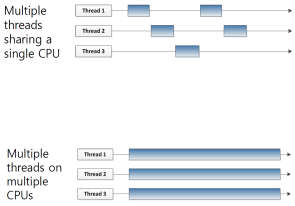
Multitasking



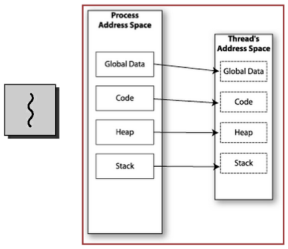
Multithreading







Memory Layout: Single-Threaded Program



Memory Layout: Multithreaded Program