CS35L

Spring 2019

Contents

CS35L

Introduction

Terms

- **Operating System**: most important software, links hardware with other processes
- *multi-user os*: allows many users to work on a single system at the same time
- *multi-process os*: running different tasks/processes at the same time
- *command line interface* vs. *graphical user interface*:
  - cli has a steeper learning curve, allows pure control, is faster, uses less resources, shows less graphical changes
- **kernel**: core of any OS, handles allocation of memory / interface of applications with hardware
  - eg. Linux is a kernel, Debian GNU/Linux is a clone of UNIX
  - controls access to system resources: memory, I/O, CPU
  - lowest layer above CPU, provides protections and fair allocations
- **shell** is the outermost layer of the kernel, interface between kernel and user
  - eg. command prompt in Windows, shell in UNIX, terminal in Mac
- the Linux file system is *hierarchal*
  - usually with single root, but allows for multiple home directories (multi-user)
- **sudo** is administrator, ie. super-user
- everything in the file system is either a process or a file

- processes are identified by a PID, files are a collection of data
- *file permissions*:
  - `ls -l` (list) prints out the following attributes, from left to right:
    - file type, user permissions, group permissions, other permissions, number of hard links, owner name, group name, size, date last modified
    - d for directory, - for file, l for link
  - eg. `drwxr-xr-x 1 someUser someGroup`
  - different permissions: read, write, executable, denied
  - can also be written in octal:
    - 3 digits: owner, group, other
    - values: execute - 1, write - 2, read - 4
      - eg. 721: full access for user, write only for group, execute only for other
  - special permissions: `setuid`, set user ID on execution
    - s flag replaces x flag for the user, users act as the owner
    - eg. `passwd` command, `chmod u+s file`
  - `setgid` grants permssion of the group owner
    - s flag replaces x flag for the group, `chmod g+s file`
- `chmod`: change mode/permissions
  - classes: u user, g group, o other, a all
  - - adds modes, - removes modes, = sets modes
  - eg. `chmod u-w file` removes write permission from the user

Character Sets and Encodings

- *ASCII* is both a character set and an encoding
  - set of 128 characters (128-255 not used), fits within a single byte
- *Unicode* is a character set with over 1,000,000 code points
- *UTF-8* is an encoding with a variable length between 1 and 4
  - *ASCII* is incorporated within *UTF-8*

Environment Variables

- variables that can be accessed from any process
  - dynamic variables
  - eg. `$HOME` to home directory and `$PATH` to search for commands
  - change with `export VARIABLE=...`
- **locale** is a set of parameters that defines a user's cultural preferences
  - default locale is C

  - eg. language, country, etc.
  - parameteres can be checked with `locale` command
  - different `$LC_*` environment variables within the locale
    * eg. `LC_TIME`, `LC_COLLATE` (differs, eg. US sort is case insensitive vs C sort)

Basic Shell Commands

- `!!` replace with previous command, `!str` refer to previous command with str
- redirection:
  - `> file` writes stdout to a file
  - `>> file` appends stdout to a file
  - `< file` use contents of file as stdin
- `cd`: change directory, `mkdir`: make directory, `rm`: removes files, use -r flag for directories or `rmdir` on empty directories
- `cp`: copy a file (src, dest), `mv`: moves a file or just renames it (src, dest)
- `pwd` prints working directory
- `ps`: processes, `du`: memory usage
- `find`: (directory) `-type -perm -name -user -maxdepth`
  - -mtime for recently modified files
  - -type d for directory, f for file, l for symbolic link
  - eg. `find . -name my*` (. searches current directory)
  - can use regex and linux wildcards
- `whatis`, `whereis` locates binary, source, and man, `which` locates binary
- `diff`: print the line differences between files, -u unified format, -p check C functions, -r recursive
- `cmp`: print the byte differences between files
- `wget` and `curl`: download files
- `man`: open manual for some command
  - -k searches for relevant manuals, apropros
- `ls`: print directory:
  - -a all files, -l list line-by-line, -t sort by modification time, -r reverse order, -i prints inode or memory references to files
  - -d list only directories, -s show size
  - can combine flags together, eg. `ls -altr`
- `ln`: link files (original, link)
  - when making a hardlink, removing the original file *does not* affect the hardlinked file
    * even though they share the *same* inode/contents

        * cannot link to a directory
        * permissions AND contents are updated
        * mirror link
    - use -s flag for soft or *symbolic* links that become *dangling* links when the original file is removed
        * the inode is *not* shared, can link to directories
        * permissions are not updated
        * actual link
- `touch`: update access and modification time to current time
    - -t set time, eg. 201101311759.30
- `echo`: prints text
    - can use redirection operators:
        * `echo TEXT > file.txt` replaces text in file with TEXT, or creates new file
        * `echo TEXT >> file.txt` appends TEXT to file, or creates new file
- `head`: prints first 10 lines of some file (analagous to `tail`)
    - can use with pipe operator:
        * `cat file.txt | head -n 2` prints the first two lines of the file
- `wc`: word count, -l for lines, -m for characters, -c for byte counts, -w for words
- `cmp`: compare
- `cat`: output contents of a file
- `ssh`: connect to server, `!ssh` calls most recent ssh command
    - `scp`: secure copy of files between servers
- `tar`: archive files
    - -x extracts, -v verbose, -f file, -c creates, –transform changes file names
- `gzip`: compress or expand files
    - -1 fast compression, -9 slowest compression
- `du` estimates file space usage, `ps` reports current processes, `kill` takes PID and kills process

Text Editing Commands

- `sort` sorts lines of text files, sort order depends on locale
    - -f ignores case, -n numeric sort, -M month sort, -r reverse, -u unique, -o output
    - defaults to ASCII sort in C locale
- `comm` compares two *sorted* files line by line
    - outputs three columns: words present in first file, second file, and both
        * supress with -1, -2, -3
    - may have to suppress to check for repeats
- `tr` translate or delete characters

  - usage: `tr [OPTION] ... SET1 [SET2]`
  - -d flag deletes characters, -s squeezes repeats, -c uses the complement of SET1
  - eg. `tr '12' 'ab'`, `tr -d [:digit:]`
    - other special sets: `[:lower:]`, `[:upper:]`
      - alpha, blank, space
- `sed` stream editor command
  - modifies the input as specified
  - `sed -n '1p'`, `sed -n '1,10p'`, `sed -n '1~2p'` prints specific lines
    - -n suppresses default print of all lines, -i edits in place instead of stream
  - can also delete text: `sed '1~2d'`, `sed '/pattern/d'`, `sed '/pattern/!d'`,
    - !d deletes inverse
  - modifying text: `sed 's/cat/dog/'`, `sed 's/cat/dog/g`, `sed 's/<[^>]*>//g'`
  - g flag global, number replaces only the numberth match of every line
  - sed -r, -E evaluates ? or `+`
- `grep` command to search files or text for the occurence that match a pattern
  - `grep -r '*.txt'`, -c gives the count, -n gives the line number, -r recursively, -i ignore case, -w whole word boundaries
  - -e specify patterns
  - -l and -L suppresses normal outputs, -v inverts sense of matching
  - -E or `egrep` to use extended regex
- `awk` is a programing language by itself, utility/language for data extraction
  - views text as fields
    - follows pattern `awk '/pattern/ {actions}' file`
  - `awk '/text/ {print;}' file.txt` prints lines which match text
  - `awk '{print $1,$2;}' file.txt` prints specific fields
  - `awk -F 'delim' '{prints $2}' file.txt` prints second column between specified pattern
  - built in variables:
    - NR count of input records, NF number of fileds, FS field separator (default whitespace), RS record separator

Regular Expressions

- regular expressions allow searching for a specific pattern
- UNIX *wildcards*: characters that can stand for all members of some class of characters

  - the * wildcard matches zero or more characters in a file or directory name
  - the ? wildcard will match exactly one character
  - the [] represents a single character matching any of the characters enclosed between them, eg. [0-9]
    * used in combination with another expression
- regex rules are slightly different...
- *quantification*:
  - how many times of the previous expression...
  - operators: ? (0 or 1), * (0 or more), + (1 or more)
  - eg. n[0-9]a matches n0a, n1a, n9a, but not na, nxa, or n10a
    * but n[0-9]*a also matches na and n10a
  - {n} exactly n times
  - {n,m} from n to m times
  - {n,} n or more times
  - quantifiers are greedy by default
  - following a quantifier with a ? makes them lazy
    * eg. \d+? matches 1 in 12345, A*? matches empty in AAA, \w{2,4}? matches ab in abcd
- *alternation*:
  - which choices...
  - operators: [] (without hyphens) and | (or operator)
  - eg. Hello | world, [A B C]
- *anchors*:
  - where...
  - characters: ^ beginning, $ end, \b word boundary
- [] match any enclosed characters, [^] match complement of enclosed character, . matches a single character of any value
  - inside character classes, only metacharacters are:] \ ^ -
    * all other special characters do not have to be escaped
    * if placed in class correctly, these do not have to be escaped either
- use \ to escape character
- regex matches whitespace as well
- *parentheses*: allow quantifiers to sequences of characters
  - eg. (a[0-9]z)* vs. a[0-9]z*
- basic regular expression vs. extended regular expression
  - BRE takes things more literally, characters such as ? or + lose their meaning
    * BRE only recognizes the metacharacters ^ $. [ ] *
  - -E means extended regular expressions in grep command
    * ERE adds the other metacharacters ( ) { } ? + |
- character classes:

  - \d digits, \D non-digits
  - \w word character, \W non-word character
  - \s whitespace character, \S non-whitespace character
- capturing groups/contents:
  - eg. (\d\d)\+(\d\d)=\2\+\1 matches 112+65=65+112
  - non-capturing group: (?:...) vs. (...)
- positive look-ahead and look-behind:
  - eg. d(?=r) matches d if it is followed by r, but r will not be part of match
  - eg. (?<=r)d matches d if it is preceded by r, but r will not be part of match
- negative look-ahead and look-behind:
  - eg. q(?!)u matches q not followed by a u
  - eg. (?<!a)b matches b not preceded by a

Shell Scripting

- in a compiled language, the code must be compiled and is not translated to machine code if there is an error
  - runs faster because it has been compiled
  - eg. C, C++, Java
- in a scripting language, there is no compilation required, it is directly interpreted/translated
  - slower
  - eg. Python, JavaScript, Shell Scripting
  - can be used for automation, less code intensive actions
- *shell script*: a program designed to be run on a shell
  - all shell commands can be executed inside a script
  - simple, portable, no compilation

Syntax

- shell recognizes built-in commands (eg. echo), shell functions, and external commands
  - eg. `echo $myvar` vs. `number=`ls | wc -l``
- first line of shell script must be a *shebang* to indicate what kind of script it is
  - when the shell runs a program, it asks the kernel to start a new process and run the given program in that process
  - `#!/bin/sh`, need to tell kernel which shell to use (eg. csh, awk, sh)
- variables in shell script can start with letter or underscore

  - typical var rules, start with letter or underscore, followed by letters, digits, or underscores
  - cannot have spaces between the variable declaration and name
  - eg. `myvar=text`, `echo $text`, `fullname="$first $middle $last"`
    * multiple assignments in same line allowed
    * need double quotes when defining *whitespace*
    * single quotes take the *literal* value of what is enclosed
      · eg. `myvar='$anothervar'`
    * usual escape character rules
- can use backticks to set a variable equal to a command
  - eg. `myvar=\`ls -l\``
- special variables: `$` PID, `#` num arguments, `n` nth argument, ? exit status of last account
- `array_name[index]=value`, `echo ${array_name[index]}`
- can redirect files using `<`
  - redirecting stderr to a file: `2>errorFile`
  - redirecting stderr to stdout: `2>&1`
  - redirecting both to file: `&>file`
- conditionals:
  - `[[ -z STRING ]]` empty string
  - `[[ -n STRING ]]` not empty string
  - `[[ STRING == STRING ]]` equality
  - `[[ STRING != STRING ]]` non-equality
  - `[[ STRING =~ STRING ]]` regex
  - `[[ NUM -eq NUM ]]` equality
  - `[[ NUM -ne NUM ]]` non-equality
  - `[[ NUM -lt NUM ]]` less than
  - `[[ NUM -le NUM ]]` less than or equal
  - same for greater than...
- file conditionals: (for current directory)
  - `[[ -e FILE ]]` exists
  - `[[ -r FILE ]]` readable
  - `[[ -h FILE ]]` symlink
  - `[[ -d FILE ]]` directory
  - `[[ -w FILE ]]` writable
  - `[[ -s FILE ]]` size greater than 0
  - `[[ -f FILE ]]` file
  - `[[ -x FILE ]]` executable
  - `[[ FILE1 -nt FILE2 ]]` newer than
  - `[[ FILE1 -ot FILE2 ]]` older than

  - [[FILE1 -ef FILE2]] equal files
- can pass arguments to a script:

```
#!/bin/bash
echo $1 $2 $3

args=("$0") # array of arguments
echo ${args[0]}

echo $#     # number of arguments
echo $0     # base arg

echo $@     # all args
echo $*     # all args, starting from first
```

- loops in a script: (similar syntax for while loops)

```
for var in list_values
  do
    commands...
  done

ALL=`ls -a $dir | sort`
declare -a ARRAY # declare syntax
count=0
for FILE in $ALL
  do
    ARRAY[$count]=$FILE # FILE is a variable
    ((count++))
    ...
  done

for i in "${ARRAY[@]}"
  do
  ...
```

```
  done
```

- conditionals:

```
if [ $a == $b ]
then
  echo "a is equal to b"
elif [ $a -gt $b ]
then
  echo "a is >= b"
elif [ $a -lt $b ]
  echo "a is <= b"
else
  echo "no conditons met"
fi

FRUIT="kiwi"
case $FRUIT in
  "apple") echo "apple pie"
  ;;
  "kiwi") echo "kiwis"
  ;;
esac

#unconditonals
break
continue
```

Software

- installing software is different for different OS's
- for Linux
  - can consolidate installation with configure, make, make install
  - software usually in the *tarball* format (.tgz or .gz)

  - other sources: rpm (Redhat Package Management), apt-get (Advanced Package Tool)
- to decompress, use `tar`
  - eg. `tar -xzvf filename.tar.gz`
  - options:
    * -x extract, -v verbose, -f file
    * -j use bzip2, -J use xz, -z use gzip or gunzip, -Z use compress
- compilation process (eg. for C++):
  1. **preprocessor** deals with preprocessor directives
  - removes comments
  - includes some header files
  - source code -> expanded source code (temporary file, can be printed on stdout)
  2. **compiler** creates the assembly level language
  - expanded source code -> assembler file (.s)
  3. **assembler** translates assembly into machine-readable code
  - assembler file -> object code file (.o)
  4. **linker** links all object and library files (specifically, their headers) together
  - object code file -> executable file
  - eg. `g++ -Wall shoppingList.cpp item.cpp shop.cpp -o shop`
    * -Wall enables all warnings
- if only a single header or source file is modified out of hundreds in a large project
  - not efficient to completely recompile every file (some files not dependent on the single modified file)
  - instead, have separate object code files for every source file, can link them all together faster
  - -c creates object files, -o flag links object files
  - also difficult to keep track of which files to recompile when the project is large
    * can use `make`

Make

- utility for managing large software projects
- helps keeps files compiled and up to date
- is a shell script with:
  - targets, dependencies, and actions
    * will try to create dependencies using other rules
    * will only recompile files whose timestamps have changed, efficient compilation

  - syntax ex. for a single rule: `target : dependencies action`
- by default, first target is run if target not specified (eg. `make`)
- by default, will output executing lines (can suppress by prepending `@` to line)
- have to escape `$` with `$$`
  - special variables:
    - `$@` current target in rule
    - `$<` first input file in rule
    - `$^` all input files (no duplicates)
    - `$?` all input files newer than the target
    - `$*` stem part matched from `%` in rule definition
    - can grab directory and file: eg. `$(@D)` and `$(@F)`
- use `.PHONY` to specify non-file rules
  - eg. `.PHONY : all clean check`
- can embed shell scripts
- each line is ran in different shell, so must escape newline for embedded scripts

```
all : shop
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
...
clean :
  rm -f item.o shoppingList.o shop.o shop

#using variables:

OPTIMIZE = -O2
CC = gcc
CFLAGS = $(OPTIMIZE) -g3 -Wall -Wextra -march=native -mtune=native -mrdrnd

randall: randall.c
    $(CC) $(CFLAGS) randall.c -o $@

#using script:
tests = 1-test.ppm 2-test.ppm 4-test.ppm 8-test.ppm
check: baseline.ppm $(tests)
```

```
    for file in $(tests); do \
      diff -u baseline.ppm $$file || exit; \
    done
$(tests): srt
    time ./srt $@ >$@.tmp
    mv $@.tmp $@
```

- build process for installing software:
  1. `configure` checks for machine details / compatibility or dependency issues
  – creates a makefile
  2. `make` uses makefile to compile program code and create executbales in current directory
  3. `make install` copies executables into final, system directories (the shell path)

Patching

- *patch*: piece of software designed to fix problems or update a computer program
- is a `diff` file that includes the changes made to a file
- diff unified format:
  – `diff -u og_file mod_file`
  – `--- path/to/og_file`
  – `+++ path/to/mod_file`
  – `@@ -l,s +l,s @@` where:
    * @@ refers to the beginning of the 'hunk'
    * l is the beginning line number
    * s is the number of lines the change hunk applies
    * lines with - sign were deleted
    * lines with + sign were added
- apply patches with `patch` command
  – eg. `patch -pnum <patch_file`
  – pnum strips off leading slashes in order to generate a relative path

Python

- not just a scripting language, also object oriented
- compiled and interpreted
- not as fast as C

- fewer syntactical constructions, English keywords instead of symbols
  - automatic garbage collection
  - interactive and script modes
  - no semicolons / braces for blocks of codes
    - instead uses indents, strictly enforced
    - blocks must be equally indented
  - however, ignores spaces within statements
- python is *loosely* typed
- python variables:
  - start with underscore or letters
  - followed by other letters, underscores, or digits
  - no reserved words
- in python2, print is a statement
  - in python3, print is a function “‘py #!/usr/bin/python counter = 100 miles = 1000.0 name = “John” print counter print miles print name

if (counter == 100): print “correct” print “good-bye”

#here is a comment """here is a multiline comment""" - python list is like a C array but is dynamically sized - can also hold objects of different types - python dictionary is a hash table - key-value pair storage capability - keys unique, values not - keys and values can be different types - to add to a list: - `my_list.append(obj)` appends an object to the end of the list - `my_list.extend(iterable)` appends each element of the iterable onto the list - for a string, extend will append chars onto the listpy list = [1, 3.14, ‘str’] list1 = [1, 2, 3, 4] list2 = [2, 3, 4, 5] list1 += list2 print list print list[0] list1[0] = 100 print list1

dict = {} dict[‘france’] = ” paris” print dict[‘france’]

if (dict[‘france’] == “paris”): print “correct” elif (dict[‘france’] == “europe”): print “maybe” else: print “wrong”

del dict[‘france’] del dict

dict = { ‘name’:‘test’, ‘class’:3 }

for i in list1: print i for i in range(len(list1)): # from 0 up to length of list print i - string operations:py s = ‘Hello’ #s[1:4] is ell #s[1:] is ello #s[:] is Hello #s[1:100] is ello

#can also use negative based indexing #s[-1] is o #s[-4] is e #s[:-3] is He #s[-3:] is llo

#can split using delimietrs x = “blue, green, red” arr = x.split(“,”) # arr is [‘blue’, ‘green’,

'red'] - functions:py def printme(some_str): print some_str return

def find_sum(some_list): sum = 0 for element in some_list: sum += element return sum - classes:py class Rectangle: def **init**(self, x, y): self.l = x; self.w = 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:", rect.getArea()) print("Perimeter:", rect.getPerimeter())

main() - can use optparse library to parse through argument, options, and argument -options when writing scripts - I/O basics: - `raw_input("string prompt")` returns a string with trailing newline removed - `input("string prompt")` assumes input is valid Python expression - to check if void user input, `if input_str:` - file I/O: - `f = open(filename, 'r')` creates a file handle - `lines = f.readlines()` creates a list of strings from handle - `f.close()` closes handle - optparse library and error handling:py parser = OptionParser(version=version_msg, usage=usage_msg) parser.add_option('-n', '–numlines', action='store', dest='numlines',) # stores next argument into destination numlines default=1, help='output NUMLINES lines (default 1)') options, args = parser.parse_args(sys.arv[1:]) # options, object with all option args, args is leftover positional args

try: numlines = int(options.numlines) except: parser.error('invalid NUMLINES: {0}'.format(options.numlines)) # catching exceptions

if ... raise ValueError('error message') # raising an error

if **name** == '**main**': main() # making Python file standalone "'

C Overview

- basic data types:
  - int, float, double, char, void
  - no bool!
  - size_t (unsigned integer)
- different ways to pass arguments / stdio to a script:
  - `cat file | ./script`
  - `./script <file`
  - `./script file` (can access using `argv` variable)

- *double pointer*:
  - `int* p = &x`, and then `int** q = &p`
  - q is a double pointer
- pointers can point to arrays, and arrays are pointers:

```
int A[5];
int* p;
p = &A[0];
p = A;
*p == A[0];
p + 1 == &A[1];
*(p + 1) == A[1];
A + 2 == &A[2];

// with 2-D arrays
int A[2][3];
A == &A[0];
A + 1 == &A[1];
**(A + 1) == A[1][0];
A[0] + 1 == &B[0][1]; // A[0] is type int*

// can use function pointers to pass a function to another
int (*fp)(int, int);
fp = &add;
sum = (*fp)(2, 3);
// or
fp = add;
sum = fp(2, 3);

// can use with library functions such as qsort
#include <stdlib.h>
void qsort(void* base, size_t num, size_t size,
           int(*compare)(const void*, const void*));
// can pass in a comparison function to qsort
// < 0 if p1 goes before, 0 if equivalent, > 0 if after
```

```c
int compare(const void* a, const void* b) {
  return *(int*)a - *(int*)b;
  // for comparing strings/letters, would have to cast from char** type:
  // char* letter = (char**) a;
}

int values[] = { 40, 30, 60 };
qsort(values, 6, sizeof(int), compare);
```

Pointer Example

```c
#include <stdio.h>
char *c[] = { "the","quick brown fox","jumped","over the","lazy dog" };
char **cp[] = { c+3, c+2, c+1, c, c+4 };
char ***cpp = cp;

int main(void)
{
  printf("%s ", *(c+1));
  printf("%s ", *(c+2));
  printf("%s ", **(cpp+3));
  printf("%s ", **(cp+4)+4);
  printf("%s ", **(++cpp));
  printf("%s ", **cp);
  printf("%s ", **(++cpp)+5);

  /* the
     _dog
     jumped
     over_the
     _brown_fox */
}
```

Structs

- no classes in C
- in structs:
  - package related data
  - no member functions
  - no access specifiers (private by default)
  - no constructors
- can use typedef as syntactic "sugar"

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

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

Student* sp = &s;
sp->age = 18;
```

Dynamic Memory

- memory that is allocated at runtime, will be allocated on the *heap*
- must know size of the array at compile time
  - for dynamic arrays, must use allocated memory
- `void* malloc(size_t size)`:
  - allocates size bytes and returns pointer to that memory
  - returns NULL if memory not allocated
  - eg. `ptr = (int*)malloc(n * sizeof(int))`
- `void* calloc(size_t num, size_t size)`:
  - allocates block of memory for array of num elements, each size bytes

- `void* realloc(void* ptr, size_t size)`:
  - changes the size of the memory block pointed to by ptr to size bytes
  - contents of memory is unchanged
- `void free(void* ptr)`:
  - frees the block of dynamic memory pointed to by ptr
  - double free has undefined behavior

I/O

- read characters from stdin with `getchar()`
- write characters to stdout with `putchar(int char)`
  - these are unbuffered input and output
- `fp = fopen("file.txt", "w+")`
- formatted I/O with `fprintf()` and `fscanf()`
  - `int fprintf(FILE* fp, const char* format, ...)`
    - fp can be file pointer, stdin, stdout, stderr
  - eg. `fprintf(stdout, "%s has %d points\n", player, score);`
- can write to stderr as well with `perror()`

Debugging

- debugger allows programmers to:
  - step through source code line by line
  - inspect program at runtime
- GDB for C
- compile with -g flag to use with debugger
  - `gdb <exe>` or `gdb` and `(gdb) file <exe>`
  - `(gdb) run` or `(gdb) run [args]`
  - `(gdb) help <command>`
- useful functionalities:
  - set breakpoints
    - `(gdb) break file.c:6`
    - `(gdb) break my_function`
    - `(gdb) break [position] if expression`
    - `(gdb) info break`
    - also delete, disable, enable, ignore breakpoints
  - check variables with `print [/format] expression`
    - formats: d, x, o, t (binary)
  - `watch` changes to variables, `rwatch` expression

  - step (steps into subroutines), next (does not trace into subroutines)
  - continue (next breakpoint), finish (until current function returns)
- process memory layout:
  - TEXT segment with instructions
  - global variables (initialized and uninitialized)
  - heap segment grows upward (dynamic memory allocation, malloc, free)
  - stack segment grows downward (push, pop, stores)
    - stack frame set aside for called functions
    - holds local variables, parameters, return address, etc.
  - (`gdb`) `backtrace` shows the call trace / stack
  - (`gdb`) `info frame` shows registers, return address of current frame
  - (`gdb`) `info locals` shows local variables and values
  - (`gdb`) `info args` shows argument values
  - (`gdb`) `info functions` shows functions
  - (`gdb`) `info list` lists source code lines around current line

System Call Programming

- processor modes place restrictions on type of operations by processes
  - **user** space vs. **supervisor** space
    - user applications and C library are in the user space
    - in user space, some memory cannot be accessed, cannot do some I/O
  - kernel/supervisor space has *unrestricted* access
    - access all areas of memory, can take over cpu
    - exception will crash the OS
    - CPU mode bit changes from 1 to 0
- this allows protection of I/O, memory, and CPU
  - prevent processes from messing with each other and the OS
- **system calls** are a special type of function:
  - making a type of kernel call from user-level processes
  - provide interface between kernel and user programs
    - only way user program can perform privileged operations
  - when system call is made:
    - program is interrupted
    - control is passed to the kernel and interrupt handler
    - also handle exceptions and *traps*
  - have an *overhead*
    - expensive and can hurt performance

        * OS must interrupt and save state of program
        * take control of CPU and verify operation
        * restore saved context, and restore CPU to user process
- in C, system calls are used the same way as calling a procedure or a function
    - but only a system call can enter the kernel
    - also provide C library functions to make system calls
        * fewer system calls, less switches in control

Types

1. process control - stopping execution of a running programing

- eg. fork, exit, wait

2. file management - open, close, read, write, create
3. device management

- read, write, ioctl (input-output control)

4. information management - transfer info between user and OS, eg. date or time

- getpid, alarm, sleep

5. communication - talk between different processes

- pipe, shmget (allocates shared memory segment), mmap (maps files or devices into memory)

Example System Calls

- file descriptor: integer that identifies open file of process
- file descriptor table: collection of integer array indices that are file descriptors
    - elements are pointers to file table entries
    - one for each process
- `int open(const char* Path, int flags)`
    - path can use relative or absolute path (starts with /)
    - flags include `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT` (create file), `O_EXCL` (prevent creation)
        * `O_APPEND` file offset set to end of file before each write
        * `O_TRUNC` truncates existing file to size 0
    - returns file descriptor used (starts at 3 since 0-2 are taken by system)
- `int creat(char* filename, mode_t mode)`
    - mode indicates permission
    - returns first unused file descriptor, generally 3

- open and close return -1 on an error
- to avoid system call overhead, can use equivalent library functions:
  - these make system calls, but have been optimized to make fewer system calls
  - `getchar()`, `putchar()`
  - `fopen()`, `fclose()` “‘c #include <fcntl.h> #include <unistd.h>

// file descriptors: 0 stdin, 1 stdout, 2 stderr // note: ssize_t can take values from -1 to Tmax

ssize_t read(int fildes, void* buf, size_t nbyte); // file descriptor, buffer to write to, num bytes to read // returns actual number of bytes read, -1 is an error, 0 is EOF

// read is unbuffered, direct system call reading one byte at a time // vs. getchar is buffered, library implemented system call, // reads multiple bytes at one time into a block (buffered usually faster)

ssize_t write(int fildes, void* buf, size_t nbyte); // file descriptor, buffer to write from, num bytes to write // returns actual number of bytes written, -1 is an error, 0 is EOF

// to read from a file: fildes = open(“foo.txt”, O_RDONLY); if (fildes < 0) { perror(“error”); exit(1); } close(fildes);

// lseek() repositions file descriptor offset off_t lseek(int fd, off_t offset, int whence) // SEEK_SET set offset // SEEK_CUR current location + offset // SEEK_END size of file + offset

int fstat(int fildes, struct stat* buf); // returns information about a file from filedes in a structure // negative if error struct stat buf; buf.st_size buf.st_nlink // number of links buf.st_ino // inode S_ISDIR(buf.st_mode) // directory? S_ISLNK(buf.st_mode) // symbolic link? buf.st_mode & S_IRUSR // u+r? buf.st_mode & S_IWUSR // u+w? buf.st_mode & S_IXUSR // u+x? ...S_IRGRP ...S_IROTH

int ret = fstat(STDIN_FILENO, &buf); // STDIN_FILENO is 0 if (S_ISREG(buf.st_mode)) printf(“Regular File”);

cat file.txt | ./fstat // non regular file, 0 bytes in size ./fstat < file // regular file, n bytes in size ./fstat < /proc/self/status // regular file, 0 bytes in size! (growing file) // can use sleep or GDB breakpoint to debug “‘ ## Parallelism ***

- **multiprocessing**: the use of multiple CPU cores to run multiple tasks simultaneously
  - uni-processing system vs. a multiprocessing one
  - on uni-processing system, uses time-sharing, switches between different threads

        * creates an illusion of parallelism
    - vs. multiple cores running the threads at the same time
- **parallelism**: executing several computations simultaneously to gain performance
    - *multitasking*: several processes scheduled alternatively or simultaneously
        * different processes set different address spaces (bad for sharing, good for protecting)
        * expensive creation/destruction
        * insulated from errors in other processes
    - *multithreading*: same job is broken logically into pieces or threads which may be executed simultaneously
        * all threads in same process share the same memory (good for sharing, bad for protecting)
            · can access each others' memory
            · light-weight threads creation, easy communication
        * creating and destroying threads is easier to do, as opposed to creating an entirely new process
        * one error can crash all the threads
    - *embarassingly parallel*: an application where threads do not have to synchronize with each other
        * good candidates for synchronization
- **thread**: a flow of instructions or a path of execution within a process
    - smallest unit of scheduling by OS
    - process *consists* of up to one thread
    - on a uni-processor, processor switches between different threads, parallelism is an *illusion*
    - on a multi-processor, multiple processors run threads at the same time, *true* parallelism
- if there are global variables or pointers being used, multiple threads can access the same variable
    - sharing the same resources
    - can lead to issues with synchronization, a *race condition*
        * eg. incrementing a value twice on two separate threads
            · may only result in a single increment if they execute at the same time
    - can solve by "locking" and "unlocking" critical sections
- critical sections:
    - *mutex*: object that allows only one thread into a critical section
        * owned by a thread, forces other threads to wait
        * each resources has a mutex

        * `pthread_mutex_t lock;`
        * `pthread_mutex_init(&lock, NULL);`
        * `pthread_mutex_lock(&lock);`
        * `pthread_mutex_unlock(&lock);`
        * `pthread_mutex_destroy(&lock);`
    - *semaphore*: value in a designated place in the OS that each process can check or change
        * signaling mechanism
        * restricts number of simultaneous threads of a shared resource up to a maximum number
        * requesting access to a resource decrements semaphore
            · signal on completion increments the semaphore
- C pthread functions found in pthread.h, use -pthread when compiling
- thread id held in `pthread_t` structure
- can create a thread in C with a function using pthread_create:
    - `int pthread_create(pthread_t* thread, attr, void* (*someFunct) (void*), void* arg)`
    - parameters:
        * address of pthread_t to create thread in
        * thread attributes (priority, stack size), usually NULL
        * function to run in thread, must return `void*` and take `void*` argument
        * `void*` argument to pass (only one argument taken, may have to use a struct)
    - on success, returns zero, otherwise an error number
    - use -lpthread option when compiling
- `int pthread_join(pthread_t thread, void** retval)` waits for another thread to complete
    - can also catch return values from the thread
    - takes thread ID and exit status
    - error if nonzero return
- `pthread_equal()` compares thread ID's to see if they are equal
- `pthread_exit()` can return a value, completely exit thread

```c
/* Passing struct to a thread */
struct Point {
  int x, y;
  struct Res* r;
};
struct Res {
```

```
  int sum;
};

void* add(void* pointPtr) {
  struct Point* myPoint = (struct Point*)pointPtr;
  for (int i = 0; i < 2; i++) {
    struct Res* myRes = myPoint->r;
    myRes->sum = myPoint->x + myPoint->y;
    myPoint++;
  }
}

int main()
{
  struct Point arr[4];
  struct Res arr1[4];
  struct Point* p1 = &arr[0];
  struct Point* p2 = &arr[2];
  ...
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, add, (void*)p1);
  pthread_create(&thread1, NULL, add, (void*)p2);
  pthread_join(thread1, NULL);
  pthread_join(thread1, NULL);
}
```

Linked Libraries

- an object code library is a previously compiled collection of standard program functions
- *static linking* links an entire file into a program
  - static libraries have .a extension
  - `gcc -static hello.c -o hello-static` (treats dynamic libraries as static)
- *dynamic linking* allows a process to add, remove, replace, or relocate object modules during its execution

  - not copying the entire library, just **referencing** the path to that library
  - thus, the linking is only **completed** at run time
    - the size *increases* while executing after incorporating libraries
  - dynamic/shared libraries have a .so extension
    - or .dll on Windows
  - some advantages:
    - more space efficient than copying the entire library
    - multiple programs can use the library
    - also will *not* have to **recompile** the libraries when they are updated
  - some disadvantages:
    - slightly slower execution
    - no control over specific version of libraries
    - unusable if the library does not exist
  - `gcc hello.c -o hello-dynamic`
- in *dynamic loading*, only functions that are used from libraries are copied at runtime
  - instead of the entire library, as with *dynamic linking*
  - even slower execution

Creating and Using Libraries

- to create a **static** library:
  - compile without linking: `gcc -c lib_mylib.c lib -o lib_mylib.o`
  - create a static library: `ar rcs lib_mylib.a lib_mylib.o` (archiver)
    - c creates, s writes index, r replaces, q quick append without replacement, v verbose
  - compile main code: `gcc -c main.c -o myMain.o`
  - link to main code: `gcc -o mainCode myMain.o -L. -l_mylib` (syntax for library path)
- to create a **shared** library:
  - create C and header file for library: `lib_mylib.c`, `lib_mylib.h`
  - compile without linking: `gcc -fPIC -c lib_mylib.c -o lib_mylib.o`
  - create dynamic library: `gcc -shared -o lib_mydynamiclib.so lib_mylib.o` (no archiver)
  - link to main code: `gcc -Wl,-rpath=$PWD -o mainCode main.c -L. -l_mydynamiclib`
    - -Wl specifies where the library comes from during runtime, specifying linker options
    - two parts:
      - compiling and outputting errors with the library (-L)

        - the relative path reference to use when linking(-Wl)
- GCC flags:
  - -fPIC outputs position independent code, required for shared libraries
  - -l*library* links with lib*library*.a
    - static libraries, or object file archives
  - -L, at compile time, find the library from this path
    - by default, checks /usr/lib
  - -Wl passes options to linker
    - -rpath at runtime finds .so from this path
  - -c generate object code from C but do not link
  - -shared produces a shared object which can then be linked with other objects to form an executable
- dynamic loading has a unique API
- to use shared library with **dynamic loading**:
  - `void* dlopen(const char* file, int mode)` makes an object file accessible to a program
    - mode flags include:
      - `RTLD_LAZY` resolve undefined symbols as code is executed
      - `RTLD_NOW` resolve all undefined symbols now (for debugging)
      - `RTLD_GLOBAL` can be or'ed with the other two, allows symbols to be accessible to other loaded libraries
    - returns a handle to be used by other DL library routines, can check for NULL
      - `void* dlhandle = dlopen(...)`
  - `void* dlsym(void* handle, const char* name)` obtains the address of a symbol within an opened object file
    - use with a function pointer, eg. `int (*myfunc)(int*) = dlysym(dl_handle, "powerOfNine");`
  - `char* dlerror()` returns a string error of the last error that occurred
    - should save into a char*, returns NULL after error first returned
  - `char* dlclose(void* handle)` closes an library file
  - compilation: `gcc -ldl -Wl,-rpath=$PWD -o myCode main.c` (-ldl similar to -lpthread)

Communication

- over the Internet, want certain guarantees when communicating:
  - **confidentiality** with encryption

  - data **integrity**
  - **authentication** of host and receiver identification
  - **authorization**, rights to resources
- encryption can be **symmetric**:
  - sender and receiver have the same key to decrypt
  - both parties need to know the keys
    - key may have to be *delivered* between parties
    - could be compromised
  - could use *Key Exchange Algorithm*, independently calculate same secret key
- or **asymmetric**:
  - each party has their *own* public-private key pair
    - if a message is encrypted with one key, it has to be decrypted with the other
  - data decrypted with private key, encrypted with public key, or vice versa
  - public key is published and known to everyone
    - anyone can encrypt, but cannot decrypt
  - more expensive to set up, for prolonged communication:
    - symmetric encryption is set up after asymmetric communication is established
      1. server sends copy of assymetric public key
      2. browser creates symmetric session key and encrpyts it with serber's public key
      3. server decrypts session key using asymettric private key to get symmetric session key
      4. can now use symmetric session key
    - called a *session* key
- can *authorize* a party using **asymmetric** encryption: (host validation)
  - encrypt random data with party's *public* key
  - that party can decrypt that data with their *private* key
  - reverse the data (for example), and re-encrypt the data with their *private* key
  - can decrypt the data with party's *public* key again
  - should be in reverse order if party is indeed authorized
- *public* keys must be *verified* and signed by a certificate authority
  - web browser will check certificates with this authority
- signatures provide for *authentication* of identities
  - an electronic stamp or seal
    - also ensures data integrity
  - can be appended to a document or separate (detached signature)

        - detached is stored separately
- steps:
    - generate a **message digest**
        - created using hashing algorithms
        - acts a summary of the message
        - a slight change in message can create a different digest
    - digest is encrypted using sender's private key
        - resulting encryption is *digital signature*
    - attach to message and send
- receiver can decrypt signature into digest with senders' public key
    - **only** the sender's public key can decrypt the signature
    - generate digest with same algorithm
    - compare digests
        - if not exactly the same, message has been tampered with
- GPG Handbook
    - `gpg [options]`
    - –gen-key generates new keys
    - –armor ASCII format
    - –export, –import
    - –detach-sign creates detached signature
    - –verify verifies signature with public key
    - –encrypt, –decrypt
    - –list-keys, –send-keys, –search-keys

SSH

- secure socket shell, used to remotely access shell
- encrypted and more authenticated than predecessors
- client ssh's to remote server
    - first time, asks for host *validation*
        - checks server's public key against saved public key
    - no central authority for servers, user must manually trust host
- *man-in-the-middle* attack is where an attacker assumes the keys of another server
    - eg. a server's keys were compromised and updated without the user knowing
    - check host public key with save public key
    - must manually remove key from the trusted host
- password-based authentication vs. key-based authentication
    - key-based:

  - generate key pair
  - private key is used to to decrpyt challenge message from server
    * private key still secured by passphrase, user still must enter passphrase
- to avoid reentering passphrase, use ssh-agent, which stores a copy of the private key in memory
  - this copy is used to answer challenge messages
  - still must enter passphrase once to load into memory
  - kernel still protects memory from being read by other processes
- `ssh-keygen` generates key-pair
  - `sudo useradd <user>`, `sudo passwd <user>`
  - `sudo mkdir .ssh`
  - `sudo chown -R <user> .ssh`, `sudo chmod 700 .ssh`
  - `ssh-copy-id -i` uses local key to authorize logins on remote machine (-i identify file only)
  - `ssh-add` uses ssh-agent
- can use `ifconfig` or `hostname -I` to get hostname and IP addressses

Sample Questions

1. a. If you don't trust your server, you cannot use OpenSSH to connect to it, as it can easily corrupt your client.
   b. If you don't trust your client, you can still use OpenSSH to connect to a trusted server.
   c. If you don't know the name or IP address of your server, you can use OpenSSH to discover this info in a secure way.

   d. If you don't know the name or IP address of your client, you can still use OpenSSH to connect to a server.

   e. If you don't trust your network, you an still use OpenSSH to discover whether your server is running or not.

2. **a. ssh-agent improves security by making a copy of private keys.**

   b. ssh-agent acts on your behalf by running on the server and executing commands there, under your direction.
   c. Even if the attacker replaces the ssh-agent program with a modified version your communications will still be secure.
   d. ssh-agent eliminates all need for password authentication when communicating to SEASnet hosts.
   e. If you successfully use ssh-agent and then log out from the client and then

log back in again, you can then connect to the same SSH server again without any additional passwords or passphrases.

3. a. OpenSSH typically uses public-key encrpytion for authentication, because private-key encryption is less secure.

b. OpenSSH typically uses private-key encryption for data communication, because public-key encrpytion is less efficient.

c. When you run ssh, it chooses its authentication key randomly from a large key space.
d. The OpenSSH client and server are essentially symmetric, so that it's easy and common to use the same program as either a client or a server.
e. Once your private keys are 1024 bits long, there's no reason to make them any longer, as they're impossible to break.

4. **a. OpenSSH is not limited to just one client-server connection. A team can use OpenSSH to communicate information to each other.**

b. For security, OpenSSH refuses to connect to programs written by other people. A client running OpenSSH code will only connect to a server running OpenSSH code.
c. Although port forwarding can be used to display from OpenSSH server to an OpenSSH client, the reverse is not possible.
d. For security, port forwarding cannot be chained.
e. When using port forwarding to connect to SEASnet, one should take care not to create a forwarding loop, leading toa cycle of packets endlessly circulating on the Internet.

5. a. It's not a good idea to connect to a SEASnet server and use GPG on the server to sign a file, because then an attacker on the network can snoop the GPG passphrase.
b. A detached signature file must be protected as securely as the private key. Otherwise, an attacker will be able to forge your signature more easily.
c. When generating a key pair it's important to use a private entropy pool. Otherwise, an attacker might be able to guess your key by inspecting the public entropy pool.

d. Exporting a GPG public key to ASCII format neither improves nor reduces its security.

e. Because a deached cleartext signature isn't encrypted, it is easily forged by an attacker with access to the file being signed.

Version Change Management

- software development process involves a lot of changes
  - new features, bug fixes, performance enhancements
  - many different versions and large software teams
- with multiple users across a server, must save file history
  - track changes to code and other file
  - **version control software**, eg. Git, Subversion, Perforce
- distributed vs. centralized version histories
- in *local VCS*, organizing different version as local folders
  - no server, hard to distribute to other users
- in *centralized VCS*, version history sits on a central server
  - changes must be committed
  - all users can get the changes
  - if central server fails, there are no *local* copies
- in a *distributed VCS*, version history is replicated at every user's machine
  - users have version control all the time
  - changes can be communicated between users
  - eg. Git

Git

- everything in Git is checksummed
  - hash values for every object, SHA-1 hashing
- terms:
  - **head**: currently active head, commit object
  - **branch**: refers to a head and its entire set of ancestor commits
  - **master**: default branch
- local operations (on local database):
  - `git add` pushes to the *staging area* (`git rm` put into staging for removal from repo)
    - selectively stage commits
    - can use `.gitignore` file to ignore filenames/patterns to ignore while staging
  - `git commit` pushes from staging area to the Git directory (repository)
    - if files in staging area are modified further, have to re-add
  - `git checkout` checks out a previous version
    - eg. `git checkout <hash>`
- steps to make a Git repository:

  - `git config`, –list, –global
  - `git init`
  - `git add somefile`
  - `git commit -m "Check-in ..."`
    * do not need to specify files, commit pushes everything in staging area to local repository
  - `git commit -a -m "Check-in ..."` automatically stages modified/deleted files
    * `git commit --amend` redoes a commit with a new message
  - `git reset` unstages files
    * eg. `git reset HEAD <file>` unstages file to current commit
- `git status` gives status of files, -s for short
  - files untracked if not added to repository yet
  - files not staged for commit if they have been changed
  - files to be committed if in staging area
- `git log` gives commit history, -p successive difference in patching
- `git diff` gives diff of files
  - diff of changes in the working directory
  - –staged compares with last commit

Branches

- a *branch* is a pointer to one of the commits in the repo (head) and all ancestor commits
  - pointer to one of commits (HEAD), and all ancestor commits
    * default master branch points to last commit made, moves forward with every commit
  - branches can create commits without changing the master branch
    * expirement code without affecting main branch
    * separate projects that once had a common code base
  - `git branch newBranch`
  - `git checkout someBranch` switches branches (on file level, detaches head to look at commit history)
    * `git checkout -b newBranch` makes a new branch
    * `git checkout -b newBranch existingBranch` bases new branch off of a previous one
  - `git merge someBranch` tries to merge multiple branches together
    * from someBranch to current branch
    * creates a new merged commit
    * can lead to *merge conflicts* which must be manually resolved
  - `git rebase` creates a linear structure of branches

        - cleaner working directory with linear commits

Remote Repositories

- remote repos are hosted on a network somewhere
  - `git remote show origin`
  - `git clone` implicitly adds origin remote
- communication with remote repository:
  - `git clone` creates a copy of an existing repository
  - `git push` can push multiple commits to the server
  - `git pull` takes changes from the central repository
    - also merges with the working directory
    - essentially `git fetch` followed by `git merge`
  - `git fetch` takes changes into the local repository
    - does NOT merge into the working directory
    - can see what changes have been made on central repository
- Git repository is stored in a linked list structure
  - commits point to previous commits
  - HEAD points to the front of the repo, indicates where new commits will go in repo

More Git Commands

- `git checkout -- <file>` discards unstaged local changes, does not change history
- `git revert` reverts commit (this creates new commits)
- `git reset` can rewrite history without making a new commit
- `git ignore` ignores extra files, such as .o
- `git clean` cleans up untracked files
- `git show <hash>` shows object in repo
- `git am` applies patches from mailbox
- `git tag` manages more human-readable tags
  - eg. `git tag -a v1.0 -m 'Version 1.0'`
  - can checkout branches from their tags as well
- `git help`

Appendix

Other Commands

- generating a random file:
  - `head --bytes=# /dev/urandom > output.txt`
- `time` command checks how long a process took
  - outputs real elapsed time (wall clock time)
  - CPU time used by the process
  - CPU time used by the system on behalf of the process
    - in the actual kernel
    - for example, memory allocation
  - CPU time is **cumulative** across threads
- `strace` intercepts and prints out system calls
  - -c gives a summary
  - -o gives an output
- `od` command formats input
  - -W specifies width
  - -c character format

Attributes of Functions

- used to declare certain things about functions in your program
  - helps compiler optimize and check code
  - C is not object oriented, but can be used when calling libraries
- placed before return type of function
- `__attribute__ ((__constructor__))` called whenever `dlopen` is called
- `__attribute__ ((destructor))` called whenever `dlclose` is called