Introduction to Linux

What is Linux?

- Open source Operating System
- Kernel: Core of OS
  - Allocates time and memory to programs
  - Handles file system and communication between software and hardware
- Shell: interface between user and kernel
  - Interprets commands user types in
  - Takes necessary action to cause commands to be carried out
- Programs

Which Linux should you use for this course?

- **Seasnet Servers**: (required for assignments)
  - lnxsrv.seas.ucla.edu
  - Username: SEAS ID
  - Password: SEAS password
  - On windows: ssh with putty, with XMing running
  - On Mac/Linux: ssh -X <username>@lnxsrv.seas.ucla.edu
- **Ubuntu Linux Distribution**
  - Debian based architecture
  - Most popular

Command Line Interface vs. Graphical User Interface

CLI	GUI
• Steep learning curve	• Intuitive
• Pure control (e.g., scripting)	• Limited Control
• Cumbersome multitasking	• Easy multitasking
• Speed: Hack away at keys	• Limited by pointing
• Convenient remote access	• Bulky remote access

Unix File System Layout

- Everything is either a **file** or a **process**
  - **Process**: an executing program
  - **File**: a collection of data
    - Document
    - Text of program written in high level language
    - Executable
    - Directory
    - Devices
  - Laid out in a **tree structured hierarchy**

The Basics: Moving Around

- **pwd** : print working directory
- **cd** : change working directory
  - **~** : home directory
  - **.** : current directory
  - **/** : root directory, or directory separator
  - **..** : parent directory

The Basics: History

- **<up arrow>**: previous command
- **<tab>**: auto-complete
- **!!**: replace with previous command
- **![*str*]**: refer to previous command with str
- **^[*str*]**: replace with command referred to as str

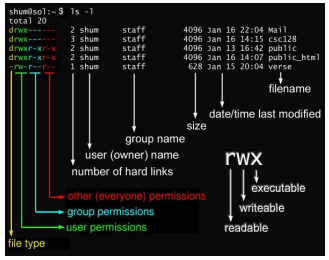
The Basics: Dealing with Files

- **mv**: move/rename a file (no undos!)
- **cp**: copy a file
- **rm**: remove a file
  - **-r**: remove directories recursively
- **mkdir**: make a directory
- **rmdir**: remove a directory
- **ls**: list contents of a directory
  - **-d**: list only directories
  - **-a**: list all files including hidden ones
  - **-l**: show long listing including permission info
  - **-s**: show size of each file, in blocks
  - **-h**: human readable form (shows size in Byte\KB\MB...)

The Basics: File Name Matching

- **?**: matches any single character in a filename
- *****: matches one or more characters in a filename
- **[]**: matches any one of the characters between the brackets. Use '-' to separate a range of consecutive characters.

The Basics: File Permissions



The Basics: Changing File Attributes

- **ln**: create a link
  - Hard links: points to physical data
  - Soft links aka symbolic links (-s): points to a file
- **touch**
  - Update access & modification time to current time
  - Can also be used to create a file
- **chmod**
  - read (r), write (w), executable (x)
  - User, group, others

The Basics: Man Pages

- **Manual (Man) pages**
  - **man**: get manual or man pages
  - **man ls**: shows the man page for 'ls' command
  - **/keyword**: forward slash followed by the word you are searching for to search within a man page
  - **q**: quit the man page

The Basics: Redirection

- **> *file***: write stdout to a file (potentially overwriting)
- **>> *file***: append stdout to a file
- **< *file***: use contents of a file as stdin

The Basics: find

- **type**: type of a file (e.g,, directory, symbolic link)
- **perm**: permission of a file
- **name**: name of a file
- **prune**: don't descend into a directory
- **ls**: list current file(s)

Shell Scripting
and
Regular Expressions

CS 35L
Spring 2018 - Lab 3

Logistics

New Office Hours (For Me):
Monday 11:30AM - 1:30PM
Boelter 2432
Posted on CCLE

Or try Email/Piazza!

Lab setup - *Locale* for Assignment 2

- Please set your Locale:
  - `export LC_ALL='C'`
- Important because we want the 'sort' shell command to be ASCII character complainant
  - Otherwise your output for 'sort' is unknown and not deterministic, and your assignment results will not be as expected

Regular Expressions

Basic I/O Redirection

- Most programs read from stdin
- Write to stdout
- Send error messages to stderr

```
$ cat                   With no arguments, read standard
                        input, write standard output
now is the time         Typed by the user
now is the time         Echoed back by cat
for all good men
for all good men
to come to the aid of their country
to come to the aid of their country
^D              Ctrl-D, End of file
```

Redirection and Pipelines

- Use *program* < *file* to make *program*'s standard input be *file*:
  `cat < file.txt`
- Use *program* > *file* to make *program*'s standard output be *file*:
  `cat < file.txt > file2.txt`
- Use *program* >> *file* to send *program*'s standard output to the end of *file*.
- Use *program1* | *program2* to make the standard output of *program1* become the standard input of *program2*.
  `cat assign2.html | tr -c 'A-Za-z' '[\n*]'`

Sorting words

- Investigate the 'sort' command
- man sort
- sort all the words in
  - /usr/share/dict/words
- save to your home folder
- sort [option] /usr/share/dict/words …?
- `sort -d /usr/share/dict/words > words`
- What does > do???

The tr command (2)

- First, download assign2.html
  - `wget http://web.cs.ucla.edu/classes/spring18/cs35L/assign/assign2.html`
- `cat assign2.html | tr -c 'A-Za-z' '[\n*]'`
- Question: What does tr do?
  - Filters everything except characters from A to Z **and** from a to z
  - 'A-Za-z' is a regular expression

Searching for Text

- grep: Uses basic regular expressions (BRE)
- egrep: Grep that uses extended regular expressions (ERE)
  - grep -E
  - egrep
  - sed -r
- Fgrep: grep matching fixed strings instead of BRE or ERE.
  - grep -F
  - fgrep

Simple grep

```
$ who                   Who is logged on
tolstoy tty1 Feb 26 10:53
tolstoy pts/0 Feb 29 10:59
tolstoy pts/1 Feb 29 10:59
tolstoy pts/2 Feb 29 11:00
tolstoy pts/3 Feb 29 11:00
tolstoy pts/4 Feb 29 11:00
austen pts/5 Feb 29 15:39 (mansfield-park.example.com)
austen pts/6 Feb 29 15:39 (mansfield-park.example.com)

$ who | grep -F austen    Where is austen logged on?
austen pts/5 Feb 29 15:39 (mansfield-park.example.com)
austen pts/6 Feb 29 15:39 (mansfield-park.example.com)
```

Regular Expressions

- Notation that lets you search for text that fits a particular criterion, such as "starts with the letter a"
- Comes in two main flavors (in linux):
  - Basic Regular Expressions (BRE)
  - Extended Regular Expressions (ERE)
- Try http://regexpal.com to test your regex
- Simple regex tutorial:
  https://www.icewarp.com/support/online_help/203030104.htm

Regular expressions

Character	BRE / ERE	Meaning in a pattern
\	Both	Usually, turn off the special meaning of the following character. Occasionally, enable a special meaning for the following character, such as for \(...\) and \{...\}.
.	Both	Match any single character except NUL. Individual programs may also disallow matching newline.
*	Both	Match any number (or none) of the single character that immediately precedes it. For EREs, the preceding character can instead be a regular expression. For example, since . (dot) means any character, **.*** means "match any number of any character." For BREs, * is not special if it's the first character of a regular expression.
^	Both	Match the following regular expression at the beginning of the line or string. BRE: special only at the beginning of a regular expression. ERE: special everywhere.

Regular Expressions (cont'd)

$	Both	Match the preceding regular expression at the end of the line or string. BRE: special only at the end of a regular expression. ERE: special everywhere.
[...]	Both	Termed a bracket expression, this matches any one of the enclosed characters. A hyphen (-) indicates a range of consecutive characters. (Caution: ranges are locale-sensitive, and thus not portable.) A circumflex (^) as the first character in the brackets reverses the sense: it matches any one character not in the list. A hyphen or close bracket (]) as the first character is treated as a member of the list. All other metacharacters are treated as members of the list (i.e., literally). Bracket expressions may contain collating symbols, equivalence classes, and character classes (described shortly).
\{*n*,*m*\}	BRE	Termed an *interval expression*, this matches a range of occurrences of the single character that immediately precedes it. \{*n*\} matches exactly n occurrences, \{*n*,\} matches at least n occurrences, and \{*n*,*m*\} matches any number of occurrences between n and m. n and m must be between 0 and RE_DUP_MAX (minimum value: 255), inclusive.
\(\)	BRE	Save the pattern enclosed between \(and \) in a special *holding space*. Up to nine subpatterns can be saved on a single pattern. The text matched by the subpatterns can be reused later in the same pattern, by the escape sequences \1 to \9. For example, **\(ab\).*\1** matches two occurrences of ab, with any number of characters in between.

Regular Expressions (cont'd)

\n	BRE	Replay the nth subpattern enclosed in \(and \) into the pattern at this point. n is a number from 1 to 9, with 1 starting on the left.
{*n*,*m*}	ERE	Just like the BRE \{*n*,*m*\} earlier, but without the backslashes in front of the braces.
+	ERE	Match one or more instances of the preceding regular expression.
?	ERE	Match zero or one instances of the preceding regular expression.
|	ERE	Match the regular expression specified before or after.
( )	ERE	Apply a match to the enclosed group of regular expressions.

Examples

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

POSIX Bracket Expressions

Class	Matching characters	Class	Matching characters
[:alnum:]	Alphanumeric characters	[:lower:]	Lowercase characters
[:alpha:]	Alphabetic characters	[:print:]	Printable characters
[:blank:]	Space and tab characters	[:punct:]	Punctuation characters
[:cntrl:]	Control characters	[:space:]	Whitespace characters
[:digit:]	Numeric characters	[:upper:]	Uppercase characters
[:graph:]	Nonspace characters	[:xdigit:]	Hexadecimal digits

Backreferences

- Match whatever an earlier part of the regular expression matched
  - Enclose a subexpression with \(and \).
  - There may be up to 9 enclosed subexpressions and may be nested
  - Use \digit, where digit is a number between 1 and 9, in a later part of the same pattern.

Pattern	Matches
\(ab\)\(cd\)[def]*\2\1	abcdcdab, abcdeeecdab, abcdddeeffcdab, ...
\(why\).*\1	A line with two occurrences of why
\([[:alpha:]_][[:alnum:]_]*\) = \1;	Simple C/C++ assignment statement

Matching Multiple Characters with One Expression

*	Match zero or more of the preceding character
\{*n*\}	Exactly n occurrences of the preceding regular expression
\{*n*,\}	At least n occurrences of the preceding regular expression
\{*n*,*m*\}	Between n and m occurrences of the preceding regular expression

Anchoring text matches

Pattern	Text matched (in bold) / Reason match fails
ABC	Characters 4, 5, and 6, in the middle: abc**ABC**abcDEF
^ABC	Match is restricted to beginning of string
def	Characters 7, 8, and 9, in the middle: abcABC**def**DEF
def$	Match is restricted to end of string
[[:upper:]]{3}	Characters 4, 5, and 6, in the middle: abc**ABC**abcDEF
[[:upper:]]{3}$	Characters 10, 11, and 12, at the end: abcDEFabc**DEF**
^[[:alpha:]]{3}	Characters 1, 2, and 3, at the beginning: **abc**ABCdefDEF

Operator Precedence (High to Low)

Operator	Meaning
[. .] [= =] [: :]	Bracket symbols for character collation
\metacharacter	Escaped metacharacters
[ ]	Bracket expressions
\(\) *\digit*	Subexpressions and backreferences
* \{ \}	Repetition of the preceding single-character regular expression
no symbol	Concatenation
^ $	Anchors

sed

- Now you can extract, but what if you want to replace parts of text?
- Use sed!

  `sed 's/`***regExpr***`/`***replText***`/'`

- Example

```
sed 's/:.*//' /etc/passwd  # Remove everything
                           # after the first colon
```

Text Processing Tools

- sort: sorts text
- wc: outputs a one-line report of lines, words, and bytes
- lpr: sends files to print queue
- head: extract top of files
- tail: extracts bottom of files

The Shell and OS

The Shell and OS

- The shell is the user's interface to the OS
- From it you run programs.
- Common shells
  - bash, zsh, csh, sh, tcsh
- Allow more complex functionality then interacting with OS directly
  - Tab complete, easy redirection

Scripting Languages Versus Compiled Languages

- Compiled Languages
  - Ex: C/C++, Java
  - Programs are translated from their original source code into object code that is executed by hardware
  - Efficient
  - Work at low level, dealing with bytes, integers, floating points, etc
- Scripting languages
  - Interpreted by program
  - Interpreter reads script code, translates it into internal form, and execute programs

Why Use a Shell Script?

- Simplicity
- Portability
- Ease of development

Example

```
$ who
george      pts/2    Dec 31 16:39 (valley-forge.example.com)
betsy       pts/3    Dec 27 11:07 (flags-r-us.example.com)
benjamin    dtlocal  Dec 27 17:55 (kites.example.com)
jhancock    pts/5    Dec 27 17:55 (:32)
Camus       pts/6    Dec 31 16:22
tolstoy     pts/14   Jan 2 06:42

$ who | wc -l            Count users
6

$ who | grep littenek    Where is littenek?
6
```

Idea

- Build a script that searches for a name
  - i.e. $who | grep userWeAreLookingFor
- Check if userWeAreLookingFor is logged in
- Let's create it!
  - create a file called `finduser`

finduser

Script:

```
#! /bin/sh
# finduser --- see named by first arguement is
  logged in
who | grep $1
```

Run it:

```
$ chmod +x finduser      Make it executable


$ ./finduser littenek
```

The #! First Line

- A shell script is just a file with shell commands.
- When the shell runs a program (e.g finduser), it asks the kernel to start a new "child process" and run the given program in that process.
- First line is used to state which "child shell" to use:
  #! /bin/csh –f
  #! /bin/awk –f
  #! /bin/sh

Ubuntu Shell Scripting

- Ubuntu 6.01+ uses by default "dash" shell which is POSIX compliant
- /bin/sh isa link to /bin/dash
- "dash" and "bash" should not have any differences in use
- Bash tutorial
  - http://linuxconfig.org/bash-scripting-tutorial

Variables

- Start with a letter or underscore and may contain any number of following letters, digits, or underscores
- Declared using =
  - `Var = 'helloworld'`
- Referenced with $
  - `echo $Var`
- Reminder - echo prints to screen
  - `man echo`
  - optional: `man printf`
    - For fancier output

Adding Variables to script files

```
#! /bin/sh
STRING=”HELLO WORLD”    #assign variable
echo $STRING            #prints the value
```

Accessing Shell Script Arguments

- Positional parameters represent a shell script’s command line arguments
- For historical reasons, enclose the number in braces if greater than 9

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

> ./argtest 1 2 3 4 5 6 7 8 9 10
```

If statements

- If statements use the test command or []
- man test
  - to see the expressions that you can create
  - e.g:

```
#!/bin/bash
if test "$#" -ne 2; then    #if number of args not equal to 2 then...
    echo "Illegal number of parameters"
else
    if [ $1 -gt $2 ]; then  #if arg $1  >= arg $2 ...
        echo "1st argument is greater than 2nd"
    else
        echo "not possible"
    fi
fi
```

If statements

```
if condition
then
     statements-if-true-1
[ elif condition
then
     statements-if-true-2
... ]
[ else
   statements-if-all-else-fails ]
fi
```

Example

```
if grep pattern myfile > /dev/null
then
   ... Pattern is there
else
   ... Pattern is not there
fi
```

case Statement

```
case $1 in
-f)
    ... Code for -f option
    ;;
-d | --directory) # long option allowed
    ... Code for -d option
    ;;
*)
    echo $1: unknown option >&2
    exit 1 # ;; is good form before `esac', but not
   required
esac
```

for Loops

Generally follows form of foreach loop, for example:

```
for i in atlbrochure*.xml
do
     echo $i
     mv $i $i.old
done
```

while and until loops

Standard syntax for while loops:

```
while condition
do
   statements
done
```

Also supports negation of condition:

```
until condition
do
   statements
done
```

break and continue

- Pretty much the same as in C/C++

Functions

- Semantically, calling a function is very similar to invoking another bash script.
- Must be defined before they can be used
- Can be done either at the top of a script or by having them in a separate file and source them with the “dot” (.) command.

Example

```
wait_for ( ) {
    until who | grep "$1" > /dev/null
    do
        sleep ${2:-30}
    done
}
```

Functions are invoked the same way a command is:

```
wait_for tolstoy    # Wait for tolstoy, check every 30s
wait_for tolstoy 60 # Wait for tolstoy, check every 60s
```

Accessing Arguments

- Positional parameters represent a shell script’s command-line arguments, or arguments to a function.
- For historical reasons, enclose the number in curly braces if it’s greater than 9.

```
echo first arg is $1
echo tenth arg is ${10}
```

Function Return

The return command serves the same function as exit and works the same way:

```
answer_the_question ( ) {
     ...
     return 42
}
```

Exit: Return value

Check exit status of last command that ran with `echo $?`

Value	Meaning
0	Command exited successfully
>0	Failure during redirection
1-125	Command exited unsuccessfully. The meanings
126	Command found, but file was not executable
127	Command not found
>128	Command died due to receiving a signal

Quotes

Preface with: `world = 42`

- Three kinds of quotes
  - Backticks: ``
    - `echo `ls $world` -> <result of ls 42>`
    - Same as: `echo $(ls $world)`
  - Double quotes: “”
    - `echo “ls $world” -> ls 42`
  - Single quotes: ‘’
    - `echo ‘ls $world’ -> ls $world`

Basic Command Searching

- $PATH variable is a list of directories in which commands are found

```
$ echo $PATH
/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin
```

Simple Execution Tracing

- To get shell to print out each command as it's execute, precede it with "+"
- You can turn execution tracing within a script by using:

  `set -x`: to turn it on

  `set +x`: to turn it off

POSIX Built-in Shell Variables

Variable	Meaning
#	Number of arguments given to current process.
@	Command-line arguments to current process. Inside double quotes, expands to individual arguments.
*	Command-line arguments to current process. Inside double quotes, expands to a single argument.
- (hyphen)	Options given to shell on invocation.
?	Exit status of previous command.
$	Process ID of shell process.
0 (zero)	The name of the shell program.
!	Process ID of last background command. Use this to save process ID numbers for later use with the wait command.
ENV	Used only by interactive shells upon invocation; the value of $ENV is parameter-expanded. The result should be a full pathname for a file to be read and executed at startup. This is an XSI requirement.
HOME	Home (login) directory.
IFS	Internal field separator; i.e., the list of characters that act as word separators. Normally set to space, tab, and newline.
LANG	Default name of current locale; overridden by the other LC_* variables.
LC_ALL	Name of current locale; overrides LANG and the other LC_* variables.
LC_COLLATE	Name of current locale for character collation (sorting) purposes.
LC_CTYPE	Name of current locale for character class determination during pattern matching.
LC_MESSAGES	Name of current language for output messages.
LINENO	Line number in script or function of the line that just ran.
NLSPATH	The location of message catalogs for messages in the language given by $LC_MESSAGES (XSI).
PATH	Search path for commands.
PPID	Process ID of parent process.
PS1	Primary command prompt string. Default is "$ ".
PS2	Prompt string for line continuations. Default is "> ".
PS4	Prompt string for execution tracing with set -x. Default is "+ ".
PWD	Current working directory.

Arithmetic Operators

Operator	Meaning	Associativity
++ --	Increment and decrement, prefix and postfix	Left to right
+ - ! ~	Unary plus and minus; logical and bitwise negation	Right to left
* / %	Multiplication, division, and remainder	Left to right
+ -	Addition and subtraction	Left to right
<< >>	Bit-shift left and right	Left to right
< <= > >=	Comparisons	Left to right
= = !=	Equal and not equal	Left to right
&	Bitwise AND	Left to right
^	Bitwise Exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND (short-circuit)	Left to right
||	Logical OR (short-circuit)	Left to right
?:	Conditional expression	Right to left
= += -= *= /= %= &= ^= <<= >>= |=	Assign ment opera tor s	Right to left

Exit: Return value

Value	Meaning
0	Command exited successfully.
> 0	Failure during redirection or word expansion (tilde, variable, command, and arithmetic expansions, as well as word splitting).
1-125	Command exited unsuccessfully. The meanings of particular exit values are defined by each individual command.
126	Command found, but file was not executable.
127	Command not found.
> 128	Command died due to receiving a signal.

More on Variables

- Read only command

```
hours_per_day=24 seconds_per_hour=3600 days_per_week=7      Assign values
readonly hours_per_day seconds_per_hour days_per_week   Make read-only
```

- Export: puts variables into the environment, which is a list of name-value pairs that is available to every running program

```
PATH=$PATH:/usr/local/bin      Update PATH
export PATH                    Export it
```

- env: used to remove variables from a program's environment or temporarily change environment variable values
- unset: remove variable and functions from the current shell

Parameter Expansion

- Process by which the shell provides the value of a variable for use in the program

```
reminder="Time to go to the dentist!" Save value in reminder
sleep 120                                Wait two minutes
echo $reminder                       Print message
```

Pattern–matching operators

path=/home/tolstoy/mem/long.file.name

Operator	Substitution
${*variable*#*pattern*}	If the pattern matches the beginning of the variable's value, delete the shortest part that matches and return the rest.
Example: ${path#*/}	Result: tolstoy/mem/long.file.name
${*variable*##*pattern*}	If the pattern matches the beginning of the variable's value, delete the longest part that matches and return the rest.
Example: ${path##*/}	Result: long.file.name
${*variable*%*pattern*}	If the pattern matches the end of the variable's value, delete the shortest part that matches and return the rest.
Example: ${path%.*}	Result: /home/tolstoy/mem/long.file
${*variable*%%*pattern*}	If the pattern matches the end of the variable's value, delete the longest part that matches and return the rest.
Example: ${path%%.*}	Result: /home/tolstoy/mem/long

String Manipulation

- ${string:position}: Extracts substring from $string at $position
- ${string:position:length} Extracts $length characters of substring $string at $position
- ${#string}: Returns the length of $string

POSIX Built-in Shell Variables

Variable	Meaning
#	Number of arguments given to current process.
@	Command-line arguments to current process. Inside double quotes, expands to individual arguments.
*	Command-line arguments to current process. Inside double quotes, expands to a single argument.
- (hyphen)	Options given to shell on invocation.
?	Exit status of previous command.
$	Process ID of shell process.
0 (zero)	The name of the shell program.
!	Process ID of last background command. Use this to save process ID numbers for later use with the wait command.
ENV	Used only by interactive shells upon invocation; the value of $ENV is parameter-expanded. The result should be a full pathname for a file to be read and executed at startup. This is an XSI requirement.
HOME	Home (login) directory.
IFS	Internal field separator; i.e., the list of characters that act as word separators. Normally set to space, tab, and newline.
LANG	Default name of current locale; overridden by the other LC_* variables.
LC_ALL	Name of current locale; overrides LANG and the other LC_* variables.
LC_COLLATE	Name of current locale for character collation (sorting) purposes.
LC_CTYPE	Name of current locale for character class determination during pattern matching.
LC_MESSAGES	Name of current language for output messages.
LINENO	Line number in script or function of the line that just ran.
NLSPATH	The location of message catalogs for messages in the language given by $LC_MESSAGES (XSI).
PATH	Search path for commands.
PPID	Process ID of parent process.
PS1	Primary command prompt string. Default is "$ ".

$IFS (Internal Field Seperator)

- This variable determines how Bash recognizes fields, or word boundaries, when it interprets character strings.
- $IFS defaults to whitespace (space, tab, and newline), but may be changed
- **echo "$IFS"** (With $IFS set to default, a blank line displays.)
- More details:
  - http://tldp.org/LDP/abs/html/internalvariables.html

Introduction to Make
and
Python Scripting

CS 35L
Spring 2018 - Lab 3

Getting Started

- Login to seasnet
- Download coreutils to a temporary directory
  - how can we download a file?
- Untar\Unzip it
  - How do you unzip a file?
  - man tar
  - *cd* into the newly created coreutils folder

Tar commands

- `tar -cvf <tarfilename.tar> <target directories>` - creates tar file.
- `tar -tvf <tarfilename.tar>` - list tar file contents
- `tar -xvf <tarfilename.tar>` - extracts tar file
- Can add `-z` flag for newer LINUX distros with gnuzip for automatic compress/decompress (.gz suffux).
- Otherwise try compress command (`.Z` suffix)
- USAGE:
  - Always create tarfile in target directory (relative file/directory names)
  - Always list tarfile before extracting (insure relative file names)
  - Always extact tarfile in target directory (relative file/directory names)
- Example:

```
tar -xzvf ~/bb-1_3a_tar.gz
```

Compiling from scratch

- Common scenario
  - You download a utility from the internet to your unix machine
  - There are no binaries, but source code and makefile is available
  - Compile and build to install it
  - Reading text files in the program folder gives clues how to install the program
    - Usually INSTALL, README, readme.txt, install.txt and so on

Compiling from scratch

- Common scenario
  - The order of compilation is usually:
    - **`./configure`**
    - **`make`**
    - **`make install`**
  - man make for more details
  - view Makefile in the programs folder for details
    - e.g. search for “install:”

Configure script

- Designed to aid in developing a program to be run on a wide number of different computers
- `configure` is application specific
  - software provides it’s own configure script
- Creates the Makefile
  - Can change default behavior with options
  - `./configure --` help for more info

Makefile and make

- We need a file that instructs make how to compile and link a program. Called a Makefile.
- The make program allows you to use macros, which are similar to variables to codify how to compile a set of source code.
  - Macros are assigned as BASH variable:
    - CFLAGS= -O -systype bsd43
    - LIBS = "-lncurses -lm -lsdl“
- Makefile is invoked with
  - make <target_name>

Standard “targets”

- People have come to expect certain targets in Makefiles. You should always browse first, but it's reasonable to expect that the targets all (or just make), install, and clean will be found
  - **`make`** - compile the default target
  - **`make all`** - should compile everything so that you can do local testing before installing things.
  - **`make install`** - should install things in the right places. But watch out that things are installed in the right place for your system.
  - **`make clean`** - should clean things up. Get rid of the executables, any temporary files, object files, etc.
- Link to more info on makefile

Makefile example

- Makefile:

```
SHELL = /bin/sh
MAKE = make
CC = g++
LIBS=
CFLAGS=-DSIGSETJMP -O

hello: main.o hello.o factorial.h
        ${CC} ${CFLAGS} -o $@  main.o hello.o ${LIBS}

clean:
        rm -f *.o
```

- Run as `make`; `make clean`

Compiling coreutils

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

Bug appears with newly built coreutils

```
[User:-)@lnxsrv07 ~/cs35L/lab3/coreutils/bin]$ ls -l /bin/bash
-rwxr-xr-x 1 root root 960376 Jul  8  2015 /bin/bash


[User:-)@lnxsrv07 ~/cs35L/lab3/coreutils/bin]$ ./ls -l /bin/bash
-rwxr-xr-x 1 root root 960376 2015-07-08 04:11 /bin/bash
```

Notice the difference between invoking ls commands above

Applying the Patch

- Read the patch bug report
  - lists.gnu.org/archive/html/bug-coreutils/2009-09/msg00410.html
- Understand what part of the code is being fixed

Applying the Patch

```
diff --git a/src/ls.c b/src/ls.c
index 1bb6873..4531b94 100644
--- a/src/ls.c
+++ b/src/ls.c
@@ -2014,7 +2014,6 @@ decode_switches (int argc, char **argv)
             break;

           case long_iso_time_style:
-          case_long_iso_time_style:
             long_time_format[0] = long_time_format[1] = "%Y-%m-%d %H:%M";
             break;

@@ -2030,13 +2029,8 @@ decode_switches (int argc, char **argv)
                formats.  If not, fall back on long-iso format.  */
             int i;
             for (i = 0; i < 2; i++)
-              {
-                char const *locale_format =
-                  dcgettext (NULL, long_time_format[i], LC_TIME);
-                if (locale_format == long_time_format[i])
-                  goto case_long_iso_time_style;
-                long_time_format[i] = locale_format;
-              }
+              long_time_format[i] =
+                dcgettext (NULL, long_time_format[i], LC_TIME);
           }
         }
       /* Note we leave %5b etc. alone so user widths/flags are honored.  */
```

Apply the Patch

- Just use an editor (eg. emacs, vim).
- Recompile it: make
- A new executable ls file is created
- Compare results between new ‘correct’ executable and ‘buggy’ installed version
- Did the patch fix the bug?

General tarball/make example

```
tar -vxzf <gzipped-tar-file>
cd <dist-dir>
./configure
make
make install
make clean
```

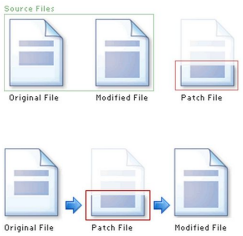
Homework

Python Scripting

Patching

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

Applying a Patch



diff Unified Format

- --- path/to/original_file
- +++ path/to/modified_file

- @@ -l,s +l,s @@
  - @@: beginning and end 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 in the new file
    - ' ' stayed the same

Patching

- `cd` into directory patch considers pwd
- vim or emacs patch_file: copy and paste the patch content
- `patch [options] [originalfile [patchfile]]`
- `patch -p`**num** `<patch_file`
- *man patch* to find out about p**num**
- BE AWARE: p**num** defaults to p1 if omitted
- `cd` into the coreutils-7.6 directory and type `make` to rebuild patched `ls.c`
- More patch command examples - link

More Python

What is Python?

- Not just a scripting language
- Object-Oriented language
  - Classes
  - Member functions
- Compiled and interpreted
  - Python code is compiled to bytecode
  - Bytecode interpreted by Python interpreter
- Not as fast as C but easy to learn, read and use

Indentation

- Python has **no braces** or keywords for code blocks
  - C delimiter: {}
  - bash delimiter:
    - then…else…fi (if statements)
    - do…done (while, for loops)
- **Indentation** makes all the difference
  - Tabs change code's meaning!!

Python List

- Common data structure in Python
- A python list is like a C array but much more:
  - **Dynamic**: expands as new items are added
  - **Heterogeneous:** can hold objects of different types
- How to access elements?
  - List_name[index]

Example

- >>> t = [123, 3.0, 'hello!']
- >>> print t[0]
  - 123
- >>> print t[1]
  - 3.0
- >>> print t[2]
  - hello!

List Operations

- >>> list1 = [1, 2, 3, 4]
- >>> list2 = [5, 6, 7, 8]
- Adding an item to a list:
  - list1.append(5)
  - Output: [1, 2, 3, 4, 5]
- Merging lists:
- >>> merged_list = list1 + list2
- >>> print merged_list
  - Output: [1, 2, 3, 4, 5, 5, 6, 7, 8]

for loops

```
list = ['Mary', 'had', 'a', 'little', 'lamb']
```

```
for item in list:
    print item
```

Result:

Mary
had
a
little
lamb

```
for i in range(len(list)):
    print i
```

Result:

0
1
2
3
4

Functions

```
def hello(strange, world="interesting"):
    print("hello " + strange)
    print(world)

hello("class")
hello(world="python",strange="everybody")
```

Argparse Library

- Powerful library for parsing command-line options (update of older optparse library)
  - **Argument:**
    - String entered on command line and passed to script
    - Elements of sys.argv[1:] (sys.argv[0] is program name)
  - **Option:**
    - An argument that supplies extra information to customize the execution of a program
  - **Option Argument:**
    - An argument that follows an option and is closely associated with it. It is consumed from the argument list when the option is

Python Walk-Through

```
#!/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 called lines
Close the file

The beginning of a function belonging to randline
Randomly select an element from self.lines and return it

The beginning of main function
version message
usage message

Python Walk-Through

```
parser = OptionParser(version=version_msg,
                      usage=usage_msg)
parser.add_option("-n", "--numlines",
    action="store", dest="numlines",
    default=1, help="output NUMLINES
    lines (default 1)")

options, args = parser.parse_args(sys.argv[1:])

try:
    numlines = int(options.numlines)
except:
    parser.error("invalid NUMLINES: {0}".
        format(options.numlines))
if numlines < 0:
    parser.error("negative count: {0}".
format(numlines))
if len(args) != 1:
    parser.error("wrong number of operands")
input_file = args[0]
try:
    generator = randline(input_file)
    for index in range(numlines):
        sys.stdout.write(generator.chooseline())
except IOError as (errno, strerror):
    parser.error("I/O error({0}): {1}".
format(errno, strerror))

if __name__ == "__main__":
    main()
```

Creates OptionParser instance

Start defining options, action "store" tells optparse to take next argument and store to the right destination which is "numlines". Set the default value of "numlines" to 1 and help message.

options: an object containing all option args
args: list of positional args leftover after parsing options
Try block
get numline from options and convert to integer
Exception handling
error message if numlines is not integer type, replace {0 } w/ input

If numlines is negative
error message

If length of args is not 1 (no file name or more than one file name)
error message
Assign the first and only argument to variable input_file
Try block
instantiate randline object with parameter input_file
for loop, iterate from 0 to numlines - 1
print the randomly chosen line
Exception handling
error message in the format of "I/O error ({errno}):strerror

In order to make the Python file a standalone program

Running Python scripts

- Download *randline.py* from assignment website
- Make sure it has executable permission:
  chmod +x randline.py
- Run it, for example
  ./randline.py –n 4 filename
  n: is an option indicating the number of lines to write
  4: is an argument to n (you can use any integer number)
  Filename: is a program argument

Python2 vs Python3

- Python2
  - First released in 2000. Final major release in 2010.
  - Considered a legacy language by many.
  - Slightly better library support (as it's older).
- Python3
  - First released in 2008. Major releases are ongoing.
  - Considered the present and future of python.
  - More limited library support, as it's newer.

For a reasonably readable rundown of the language differences, see this blog post.

Homework 3 - Overview

- randline.py script
  - Get some familiarity with python by reading the script.
  - Answer a few questions about it.
- Implement the **`shuf`** command in python
- Port your `shuf` script from python2 to 3

shuf.py

- Support the following options
  - --echo (-e), --head-count (-n), --repeat (-r), and --help
- Support variable number and types of arguments
  - File names and – for stdin
- Change usage message to describe behavior
- Port `shuf.py` to Python 3
- `man shuf` or online docs for more details
- Read coreutils shuf source if you're confused

Homework 3 Hints

- Read first 9 chapters here: docs.python.org/3.6/tutorial/
- Q4: Python 3 vs. Python 2
  - Look up "automatic tuple unpacking"
- Use `python` in shell for Python 2
- Use `python3` in shell for Python 3

```
which python3  →  /usr/local/cs/bin/python3
```

Types

- Subset of C++ (very similar)
- Built-in types:
  - Integers, Floating-point, character strings
  - No bool, false is 0 and true is anything else
- Compiling 'C' only
  - gcc -std=c99 binsortu.c

Types

- No classes, but we have **structs**
- No methods and access modifiers in C structs

```
struct Song
{
    char title[64];
    char artist[32];
    char composer[32];
    short duration;
    struct Date published;
};
```

Declaring Pointers

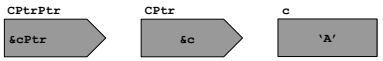
```
int *iPtr;    //Declare iPtr as a pointer to int


iPtr = &iVar; //Let iPtr point to the variable iVar


int iVar = 77;     // Define an int variable
int *iPtr = &iVar; // Define a pointer to to it
```

Dereferencing Pointers

`double x, y, *ptr;`	Two double variables and a pointer to double.
`ptr = &x;`	Let ptr point to x.
`*ptr = 7.8;`	Assign the value 7.8 to the variable x.
`*ptr *= 2.5;`	Multiply x by 2.5.
`y = *ptr + 0.5;`	Assign y the result of the addition x + 0.5.

Pointer to Pointers

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



Pointers to Functions

```
double (*funcPtr)(double, double);
double result;

// Let funcPtr point to the function pow( ).
// The expression *funcPtr now yields the
// function pow( ).
funcPtr = pow;

// Call the function referenced by funcPtr.
result = (*funcPtr)( 1.5, 2.0 );

// The same function call.
result = funcPtr( 1.5, 2.0 );
```

typedef Declarations

Easy way to use types with complex names

```
typedef struct Point { double x, y; } Point_t;

typedef struct {
   Point_t top_left;
   Point_t bottom_right;
} Rectangle_t;

typedef double banana;
banana yellow = 32.0;
```

Dynamic Memory Management

- **malloc(size_t size)**: allocates a block of memory whose size is at least *size*.
- **free(void *ptr)**: frees the block pointed to by ptr
- **realloc(void *ptr, size_t newSize)**: Resizes allocated block

```
Rectangle_t *ptr =
(Rectangle_t*)malloc(sizeof(Rectangle_t));

if(ptr == NULL) {
    printf("Malloc failed!");
    exit(-1);
}
else {
    //Perform tasks with the memory
    free(ptr);
    ptr = NULL;
}

ptr = (Rectangle_t*)
    realloc(ptr, 3*sizeof(Rectangle_t))
```

Opening & Closing Files

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

Common Streams and their file pointers

Standard input: `stdin`
Standard output: `stdout`
Standard error: `stderr`

Reading Characters

- Reading/Writing characters
  - `getc( FILE *fp );`
  - `putc(int c, FILE *pf);`
- Reading/Writing Lines
  - `char* fgets( char *buf, int n, FILE *fp);`
  - `int fputs( const char *s, FILE *fp);`

FormattedOutput

- Formatted Output to stdout
  - `int printf(const char * restrict format, ... );`
- The format string

```
double score = 42.5;
int player_number = 26;
char player[ ] = "Mary";

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

// Outputs: Mary (#26) has 42.5 points.
```

- Format Specifiers
  - %s → Strings
  - %d → Decimal integers
  - %f → floating points

Formatted Input/Output

- Formatted Input/Output For Files (including stdin/stdout)
  - `int fprintf( FILE * restrict fp, const char * restrict format, ... );`
  - `int fscanf( FILE * restrict fp, const char * restrict format, ... );`
- The format string

```
int score = 120;
char player[ ] = "Mary";
FILE* fPtr = fopen("blah", "rw");

fprintf(stdout, "%s has %d points\n", player, score);
fprintf(fPtr, "%s has %d points.\n", player, score);
// Outputs: Mary has 120 points.

fscanf(stdin, "%d", &score);
fscanf(fPtr, "%d", &score);
// Read in a single integer number and write it into score
```

Ternary Operator ?:

- Short form for a conditional assignment:

```
result = a > b ? x : y;
```

- Equivalent to:

```
if(a > b) {
  result = x;
}
else {
  result = y;
}
```

Sample Program

```
#include <stdio.h>

// Method definition before use
  void printHelloWorld();

int main(char[] argv) {
  printHelloWorld();
  return 0;
}

void printHelloWorld() {
   printf("%s\n", "Hello World!");
}
```

Compiling

- `gcc -o FooBarBinary -g foobar.c`
- The `-o` option indicates the name of the binary/program to be generated
- The `-g` option indicates to include symbol and source-line info for debugging
- For more info, `man gcc`

GDB - Debugging

Debugging Process

- Notice a bug - “Huh, that’s weird.”
- Reproduce the bug - “Well, that’s bad.”
- Simplify program input - “Is it that simple?”
- Try (and probably fail) to find bug by eye
  - “Where the $%^# is it?”
- Use a debugger to isolate problem
  - “Aha! That’s why . . .”
- Fix the problem - “And now it’s fixed.”

Debugger

- A program that is used to run and debug other (target) programs
- Advantages:

  Programmer can:
  - step through source code line by line
    - each line is executed on demand
  - interact with and inspect program at run-time
  - If program crashes, the debugger outputs where and what happened when it crashed

GDB – GNU Debugger

- Debugger for several languages
  - C, C++, Java, Objective-C… more
- Allows you to inspect what the program is doing at a certain point during execution
- Logical errors and segmentation faults are easier to find with the help of gdb

Process Layout

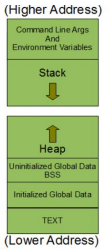


Image source : thegeekstuff.com

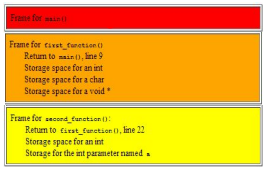
- TEXT segment
  - Contains machine instructions to be executed
- Global Variables
  - Initialized
  - Uninitialized
- Heap segment
  - Dynamic memory allocation
  - malloc, free
- Stack segment
  - Push frame: Function invoked
  - Pop frame: Function returned
  - Stores
    - Local variables
    - Return address, registers, etc
- Command Line arguments and Environment Variables

Stack Info

- A program is made up of one or more functions which interact by calling each other
- Every time a function is called, an area of memory is set aside for it. This area of memory is called a **stack frame** and holds the following crucial info:
  - storage space for all the local variables
  - the memory address to return to when the called function returns
  - the arguments, or parameters, of the called function
- Each function call gets its own stack frame. Collectively, all the stack frames make up the **call stack**

Stack Frames and the Stack

```
1  #include <stdio.h>
2  void first_function(void);
3  void second_function(int);
4
5  int main(void)
6  {
7     printf("hello world\n");
8     first_function();
9     printf("goodbye goodbye\n");
10
11    return 0;
12 }
13
14
15 void first_function(void)
16 {
17    int imidate = 3;
18    char broiled = 'c';
19    void *where_prohibited = NULL;
20
21    second_function(imidate);
22    imidate = 10;
23 }
24
25
26 void second_function(int a)
27 {
28    int b = a;
29 }
```



When main() returns, the program ends

execution within second_function()

Displaying Source Code

- list *filename*: *line_number*
  - Displays source code centered around the line with the specified line number.
- list *line_number*
  - If you do not specify a source filename, then the lines displayed are those in the current source file.
- list *from*,[*to*]
  - Displays the specified range of the source code. The *from* and *to* arguments can be either line numbers or function names. If you do not specify a *to* argument, list displays the default number of lines beginning at *from*.
- list *function_name*
  - Displays source code centered around the line in which the specified function begins.
- list, l
  - The list command with no arguments displays more lines of source code following those presented by the last list command. If another command executed since the last list command also displayed a line of source code, then the new list command displays lines centered around the line displayed by that command.

Breakpoints

- break [*filename*:] *line_number*
  - Sets a breakpoint at the specified line in the current source file, or in the source file *filename*, if specified.
- break *function*
  - Sets a breakpoint at the first line of the specified function.
- break
  - Sets a breakpoint at the next statement to be executed. In other words, the program flow will be automatically interrupted the next time it reaches the point where it is now.

Deleting, Disabling, and Ignoring BP

- delete [*bp_number* | *range*]
- d [*bp_number* | *range*]
  - Deletes the specified breakpoint or range of breakpoints. A delete command with no argument deletes all the breakpoints that have been defined. GDB prompts you for confirmation before carrying out such a sweeping command:
  - (gdb) **d** Delete all breakpoints? (y or n)
- disable [*bp_number* | *range*]
  - Temporarily deactivates a breakpoint or a range of breakpoints. If you don't specify any argument, this command affects all breakpoints. It is often more practical to disable breakpoints temporarily than to delete them. GDB retains the information about the positions and conditions of disabled breakpoints so that you can easily reactivate them.
- enable [*bp_number* | *range*]
  - Restores disabled breakpoints. If you don't specify any argument, this command affects all disabled breakpoints.
- ignore *bp_number iterations*
  - Instructs GDB to pass over a breakpoint without stopping a certain number of times. The ignore command takes two arguments: the number of a breakpoint, and the number of times you want it to be passed over.

Conditional Breakpoints

- break [*position*] if *expression*

```
(gdb) s
27 for ( i = 1; i <= limit ; ++i )
(gdb) break 28 if i == limit - 1

Breakpoint 1 at 0x4010e7: file gdb_test.c, line 28.
```

Resuming Execution After a Break

- continue [*passes*] , c [*passes*]
  - Allows the program to run until it reaches another breakpoint, or until it exits if it doesn't encounter any further breakpoints. The *passes* argument is a number that indicates how many times you want to allow the program to run past the present breakpoint before GDB stops it again. This is especially useful if the program is currently stopped at a breakpoint within a loop. See also the ignore command, described in the previous section "Working with Breakpoints."
- step [*lines*] , s [*lines*]
  - Executes the current line of the program, and stops the program again before executing the line that follows. The step command accepts an optional argument, which is a positive number of source code lines to be executed before GDB interrupts the program again. However, GDB stops the program earlier if it encounters a breakpoint before executing the specified number of lines. If any line executed contains a function call, step proceeds to the first line of the function body, provided that the function has been compiled with the necessary symbol and line number information for debugging.
- next [*lines*] , n [*lines*]
  - Works the same way as step, except that next executes function calls without stopping before the function returns, even if the necessary debugging information is present to step through the function.
- finish
  - To resume execution until the current function returns, use the finish command. The finish command allows program execution to continue through the body of the current function, and stops it again immediately after the program flow returns to the function's caller. At that point, GDB displays the function's return value in addition to the line containing the next statement.

Analyzing the Stack

- Bt
  - Shows the call trace
- info frame
  - Displays information about the current stack frame, including its return address and saved register values.
- info locals
  - Lists the local variables of the function corresponding to the stack frame, with their current values.
- info args
  - List the argument values of the corresponding function call.

Displaying Data

- p [/*format*] [*expression*]
- Output Formats
  - d: Decimal notation. This is the default format for integer expressions.
  - u: Decimal notation. The value is interpreted as an unsigned integer type.
  - x: Hexadecimal notation.
  - o: Octal notation.
  - t: Binary notation. Do not confuse this with the x command's option b for "byte," described in the next subsection.
  - c: Character, displayed together with the character code in decimal notation.

Watchpoints

- watch *expression*
  - The debugger stops the program when the value of *expression* changes.
- rwatch *expression*
  - The debugger stops the program whenever the program reads the value of any object involved in the evaluation of *expression*.
- awatch *expression*
  - The debugger stops the program whenever the program reads or modifies the value of any object involved in the evaluation of *expression*.

Using GDB

1. **Compile Program**
   - Normally: `$ gcc [flags] <source files> -o <output file>`
   - Debugging: `$ gcc [other flags] -g <source files> -o <output file>`
     - enables built-in debugging support
2. **Specify Program to Debug**
   - `$ gdb <executable>`

     or
   - `$ gdb`
   - `(gdb) file <executable>`

Using GDB

3. **Run Program**
   - `(gdb) run` or
   - `(gdb) run [arguments]`
4. **In GDB Interactive Shell**
   - Tab to Autocomplete, up-down arrows to recall history
   - `help [command]` to get more info about a command
5. **Exit the gdb Debugger**
   - `(gdb) quit`

Lab clarification

- You must specify which ls to use. Only the coreutils-with-bug version of ls will demonstrate the bug.
- "Try to reproduce the problem in your home directory, instead of the $tmp directory. How well does SEASnet do?"
  - Timestamps represented as seconds since Unix Epoch
    - Seconds or nanoseconds elapsed since January 1st 00:00 1970
  - SEASnet NFS filesystem has **unsigned** time stamps
  - Local File System on Linux server (in tmp) has **signed** time stamps
  - If you touch the files on the NFS filesystem it will return timestamp around 2054

Pointers review

- Variables that store memory addresses
- **Declaration:** <variable_type> *<name>;
  - int *ptr; //declare ptr as a pointer to int
  - int var = 77; // define an int variable
  - ptr = &var; // let ptr point to the variable var

(De)Referencing

- **Referencing**: get the address of a variable
- **Dereferencing**: getting the value that the pointer is currently pointing to
- Example:
  - double x, *ptr;
  - ptr = **&**x; //referencing: let ptr point to x
  - *****ptr = 7.8; //dereferencing: assign 7.8 to x

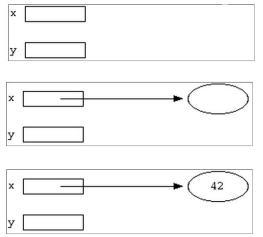
Pointer Example

int *x;

int *y;

int var; x = &var;

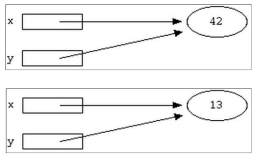
*x = 42;



Pointer Example

y = x;

*x = 13; or
*y = 13;



Pointers to Functions

- Also known as: **function pointers** or **functors**
- Goal: write a sorting function
  - Has to work for ascending and descending sorting order + other
- How?
  - Write multiple functions
  - Provide a flag as an argument to the function
  - Polymorphism and virtual functions
  - Use function pointers!!

Pointers to Functions

- Declaration
  double (*func_ptr) (double, double);
  func_ptr = &pow;
  func_ptr = pow;
- Usage:
  double result = (*func_ptr)(1.5, 2.0);
  double result = func_ptr(1.5, 2.0);

qsort Example

```
int compare (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}

int main () {
   int values[] = { 40, 10, 100, 90, 20, 25 };
   qsort (values, 6, sizeof(int), compare);
   int n;
   for (n = 0; n < 6; n++)
      printf ("%d ",values[n]);
   return 0;
}
```

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*

Valgrind

- Powerful dynamic analysis tool
- Useful to detect memory leaks

Example:

```
$ valgrind --leak-check=full
     ./sfrob < foo.txt
```

```
88 (...) bytes in 1 blocks are definitely lost ...
   at 0x........: malloc (vg_replace_malloc.c:...)
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:25)
```

Homework 4

- Implement a C function **`frobcmp`**
  - Takes two arguments **`a`** and **`b`** as input
  - Each argument is of type char const *
  - **`a`**,**`b`** point to array of non-space bytes
  - Returns an int result that is:
    - Negative if: **`a < b`**
    - Zero if: **`a == b`**
    - Positive if: **`a > b`**
    - Where each comparison is a lexicographic comparison of the unforbnicated bytes

Homework 4

- Then, write a C program called *sfrob*
  - Reads stdin byte-by-byte **(getchar)**
    - Consists of records that are newline-delimited
    - Read until end of file
  - Each byte is frobnicated
    - frobnicated - bitwise XOR (^) with dec 42
    - Sort records without decoding **(qsort, frobcmp)**
    - Output in frobnicated text to stdout **(fprintf, putchar)**
  - Dynamic memory allocation **(malloc, realloc, free)**
  - Program should work on empty and large files too

Example 1

- $ cat 'sybjre obl' > foo.txt
- Input: contents of foo.txt
  - $ ./sfrob < foo.txt
- Read the strings from stdin: sybjre, obl
- Compare strings using *frobcmp* function
- Use *frobcmp* as compare function in *qsort*
- Output: obl sybjre

Example 2

- Input: printf 'sybjre obl'
  - $ printf 'sybjre obl ' | ./sfrob
- Read the strings from stdin: sybjre, obl
- Compare and sort as in example 1
- Output: obl sybjre

Homework Hints

- Assignment 5 **requires** having a solid handle on assignment 4, so this is important!
- Use *exit*, not *return* when exiting with error
- Consider: 1-D vs. 2-D array(s)
- Test output with `od -c` or `od -a` (man od)
- Your code must do thorough error checking, and print an appropriate message on errors.
- Plug all memory leaks! (I'll be checking . . .)

System Call Programming

Low Level Process Safety

Questions:

- How do we protect processes from breaking each other? What about breaking the OS?
- Should every process be allowed to execute any command?
- How do we decide what processes deserve which permissions?

Processor Modes

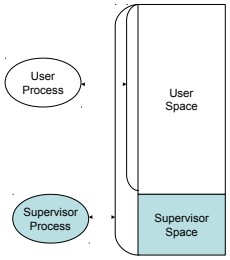
- Mode bit used to distinguish between execution on behalf of OS & behalf of user
- **Supervisor mode**: processor executes every instruction in it's hardware repertoire
- **User mode**: can only use a subset of instructions

Processor Modes

- Instructions can be executed in supervisor mode are supervisor privileges, or protection instruction
  - **I/O instructions** are protected. If an application needs to do I/O, it needs to get the OS to do it on it's behalf
  - Instructions that can change the **protection state** of the system are privileges (e.g. process' authorization status, pointers to resources, etc)

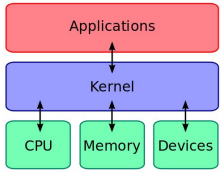
Processor Modes

- Mode bit may define areas of memory to be used when the processor is in supervisor mode vs user mode



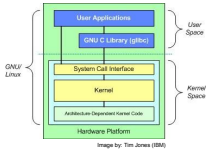
What About User Processes?

- The **kernel** executes privileged operations on behalf of untrusted user processes



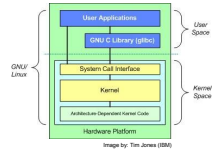
The Kernel

- Code of the OS **executing** in **supervisor** state
- Trusted software:
  - manages hardware resources (CPU, memory, and I/O)
  - Implements protection mechanisms that could not be changed through actions of untrusted software in user space

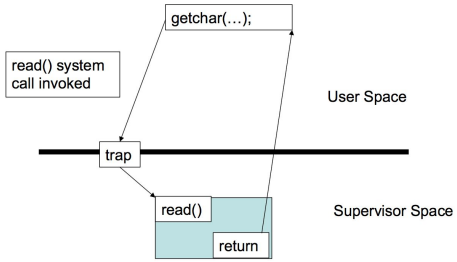


The Kernel

System call interface is a **safe way** to expose privileged functionality and services of the processor



System Calls



Trap: System call causes a switch from user mode to kernel mode

System calls

- A system call involves the following
  - The system call causes a 'trap' that interrupts the execution of the user process (user mode)
  - The kernel takes control of the processor (kernel mode\privilege switch)
  - The kernel executes the system call on behalf of the user process
  - The user process gets back control of the processor (user mode\privilege switch)
- System calls have to be used **with care**.
- Expensive due to **privilege switching**

System calls

- `ssize_t read(int fildes, void *buf, size_t nbyte)`
  - fildes: file descriptor
  - buf: buffer to write to
  - nbyte: number of bytes to read
- `ssize_t write(int fildes,const void *buf,size_t nbyte)`
  - fildes: file descriptor
  - buf: buffer to write to
  - nbyte: number of bytes to write
- `int open(const char *pathname,int flags,mode_t mode)`
- `int close(int fd)`
- File descriptors:
  - 0 stdin
  - 1 stdout
  - 2 stderr
- *Why are these system calls and not just regular library functions?*

More examples: System calls

- `pid_t getpid(void)`
  - returns the process id of the calling process
- `int dup(int fd)`
  - Duplicates a file descriptor fd. Returns a second file descriptor that points to the same file table entry as fd does.
- `int fstat(int filedes, struct stat *buf)`
  - Returns information about the file with the descriptor filedes to buf

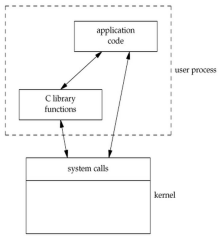
More examples: System calls

```
struct stat {
dev_t     st_dev;       /* ID of device containing file */
ino_t     st_ino;       /* inode number */
mode_t    st_mode;      /* protection */
nlink_t   st_nlink;     /* number of hard links */
uid_t     st_uid;       /* user ID of owner */
gid_t     st_gid;       /* group ID of owner */
dev_t     st_rdev;      /* device ID (if special file) */
off_t     st_size;      /* total size, in bytes */
blksize_t st_blksize;   /* blocksize for filesystem I/O */
blkcnt_t  st_blocks;    /* number of 512B blocks allocated */

time_t st_atime;  /* time of last access */
time_t st_mtime;  /* time of last modification */
time_t st_ctime;  /* time of last status change */
};
```

Library Functions

- Functions that are a part of standard C library
- To avoid system call overhead use equivalent library functions
  - getchar, putchar vs. read, write (for standard I/O)
  - fopen, fclose vs. open, close (for file I/O), etc.
- How do these functions perform privileged operations?
  - They make system calls

So What's the Point?



- Many library functions invoke system calls indirectly
- So why use library calls?
- Usually equivalent library functions make fewer system calls
- non-frequent switches from user mode to kernel mode => less overhead

Buffering issues

- What is buffering?
- Why do we buffer?
- Can we make our buffer really big?
- What happens if our program exits with a non-empty buffer?

Unbuffered vs. Buffered I/O

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

Takeaway: Buffered I/O decreases the number of read/write system calls and the corresponding overhead

Unbuffered vs. Buffered I/O examples

- **Buffered** output improves I/O performance and can reduce system calls.
- **Unbuffered** output when you want to ensure that the output has been written before continuing.
  - **stderr** under a C runtime library is *unbuffered* by default. Errors are infrequent, but we want to know about them immediately.
  - **stdout** is *buffered* because it's assumed there will be far more data going through it, and more urgent input can use stderr.
  - **logging**: log messages of a process?

Laboratory

- Write `tr2b` and `tr2u` C programs that transliterate bytes. They take two arguments 'from' and 'to'. The programs will transliterate every byte in 'from' to corresponding byte in 'to'.
  - `./tr2b 'abcd' 'wxyz' < bigfile.txt`
    - Replace 'a' with 'w', 'b' with 'x', etc
  - `./tr2b 'mno' 'pqr' < bigfile.txt`
- `tr2b` uses **getchar** and **putchar** to read from STDIN and write to STDOUT.
- `tr2u` uses **read** and **write** to read and write each byte, instead of using getchar and putchar. The nbyte argument should be 1 so it reads/writes a single byte at a time.
- Test it on a big file with 5000000 bytes

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

`time` and `strace`

- **time** [*options*] *command* [*arguments...*]
- Output:
  - real 0m4.866s: elapsed time as read from a wall clock
  - user 0m0.001s: the CPU time used by your process
  - sys 0m0.021s: the CPU time used by the system on behalf of your process
- **strace**: intercepts and prints out system calls to stderr or to an output file
  - $ strace –o strace_output ./tr2b 'AB' 'XY' < input.txt
  - $ strace –o strace_output2 ./tr2u 'AB' 'XY' < input.txt

Pointers on System Calls

www.cs.uregina.ca/Links/class-info/330/SystemCall_IO/SystemCall_IO.html

courses.engr.illinois.edu/cs241/sp2009/Lectures/04-syscalls.pdf

www.bottomupcs.com/system_calls.xhtml

Homework 5

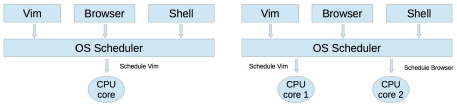
- Rewrite `sfrob` using system calls (`sfrobu`)
- `sfrobu` should behave like `sfrob` except:
  - If stdin is a regular file, it should initially allocate enough memory to hold all data in the file all at once
  - Consider outputting the number of comparisons performed
- Necessary system calls: `read`, `write`, and `fstat` (read the man pages)
- Measure differences in performance between `sfrob` and `sfrobu` using the **`time`** command
- Estimate the number of comparisons as a function of the number of input lines provided to `sfrobu`

Homework 5

- Write a shell script "`sfrobs`" that uses `tr` and the `sort` utility to perform the same overall operation as `sfrob`
- Encrypted input
  -> tr (decrypt)
  -> sort (sort decrypted text)
  -> tr (encrypt)
  -> encrypted output

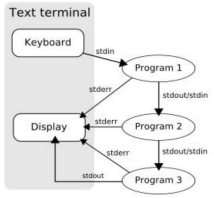
Multitasking

- Run multiple processes **simultaneously** to increase performance
- Processes do not share internal structures (stacks,globals,etc)
  - Communicate via **IPC** (inter-process communication) methods
    - Pipes, sockets, signals, message queues
- **Single core: Illusion** of parallelism by switching processes quickly (**time-sharing**). Why is illusion good?
- **Multi-core: True** parallelism. Multiple processes execute **concurrently** on different CPU cores



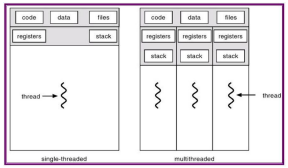
Multitasking

- `tr -s '[:space:]' '\n' | sort -u | comm -23 - words`
- Three separate processes spawned simultaneously
  - P1 - tr
  - P2 - sort
  - P3 - comm
- Common buffers (pipes) exist between 2 processes for communication
  - 'tr' writes its stdout to a buffer that is read by 'sort'
  - 'sort' can execute, as and when data is available in the buffer
  - Similarly, a buffer is used for communicating between 'sort' and 'comm



Threads

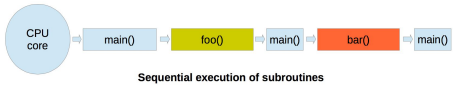
- A process can be
  - Single-threaded
  - Multi-threaded
- Threads in a process can run in parallel
- A thread is a lightweight process
- It is a basic unit of CPU utilization
- Each thread has its own:
  - Stack
  - Registers
  - Thread ID
- Each thread shares the following with other threads belonging to the same process
  - Code
  - Global Data
  - OS resources (files,I/O)



Single threaded execution

```
int global_counter = 0
int main()
{
    …
    foo(arg1,arg2);
    bar(arg3,arg4,arg5);
    …
    return 0;
}
```

```
void foo(arg1,arg2)
{
    //code for foo
}
void bar(arg3,arg4,arg5)
{
    //code for bar
}
```

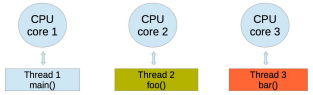


Sequential execution of subroutines

Multi threaded execution (multiple cores)

```
int global_counter = 0
int main()
{
    …
    foo(arg1,arg2);
    bar(arg3,arg4,arg5);
    …
    return 0;
}
```

```
void foo(arg1,arg2)
{
    //code for foo
}
void bar(arg3,arg4,arg5)
{
    //code for bar
}
```

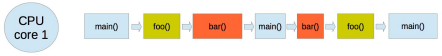


True multithreaded parallelism

Multi threaded execution (single core)

```
int global_counter = 0
int main()
{
    …
    foo(arg1,arg2);
    bar(arg3,arg4,arg5);
    …
    return 0;
}
```

```
void foo(arg1,arg2)
{
    //code for foo
}
void bar(arg3,arg4,arg5)
{
    //code for bar
}
```



Time Sharing – Illusion of multithreaded parallelism
(Thread switching has less overhead compared to process switching)

Multithreading properties

- Efficient way to **parallelize** tasks
- **Thread switches are less expensive** compared to process switches (context switching)
- Inter-thread communication is easy, via **shared global** data
- Need **synchronization** among threads accessing same data

Pthread API

```
#include <pthread.h>
```

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void* (*thread_function) (void*), void *arg);`
  - Returns 0 on success, otherwise returns non-zero number
- `void pthread_exit(void *retval);`
- `int pthread_join(pthread_t thread, void **retval);`
  - Returns 0 on success, otherwise returns non zero error number

```
#include<pthread.h>  //Compile the following code as - gcc main.c -lpthread
#include<stdio.h>

void* ThreadFunction(void *arg) {
    long tID = (long)arg;
    printf("Inside thread function with"
           "ID = %ld\n", tID);
    pthread_exit(0);
}

void CreateThreads(const int nthreads,
                   pthread_t* threadID) {
    for(long t = 0; t < nthreads; ++t) {
        int rs = pthread_create(&threadID[t], 0,
                    ThreadFunction, (void*)t);
        if(rs)
            fprintf(stderr,"Creation error\n");
            exit(1);
        }
    }
    printf("Finished creating threads\n");
}
```

```
void JoinThreads(const int nthreads,
                 pthread_t* threadID) {
    for(long t = 0; t < nthreads; ++t) {
        void *retVal;
        int rs = pthread_join(threadID[t],
                              &retVal);
        if(rs) {
            fprintf(stderr, "Joining error\n");
            exit(1);
        }
    }
}

int main(int argc, char *argv[]) {
    const int nthreads = 5;
    pthread_t threadID[nthreads];

    CreateThreads(nthreads, threadID);
    JoinThreads(nthreads, threadID);

    printf("Main thread finished execution!\n");
    return 0;
}
```

Pthread API

Thread safety/synchronization

- **Thread safe function** - safe to be called by multiple threads at the same time. Function is free of 'race conditions' when called by multiple threads simultaneously.
- **Race condition** - the output depends on the order of execution
  - Shared data changed by 2 threads
    - int balance = 1000
  - Thread 1
    - T1 - read balance
    - T1 - Deduct 50 from balance
    - T1 - update balance with new value
  - Thread 2
    - T2 - read balance
    - T2 - add 150 to balance
    - T2 - update balance with new value

Thread safety/synchronization

- Order 1
  - balance = 1000
  - T1 - Read balance (1000)
  - T1 - Deduct 50
    - 950 in temporary result
  - T2 - read balance (1000)
  - T1 - update balance
    - balance is 950 at this point
  - T2 - add 150 to balance
    - 1150 in temporary result
  - T2 - update balance
    - balance is 1150 at this point
  - **The final value of balance is 1150**

- Order 2
  - balance = 1000
  - T1 - read balance (1000)
  - T2 - read balance (1000)
  - T2 - add 150 to balance
    - 1150 in temporary result
  - T1 - Deduct 50
    - 950 in temporary result
  - T2 - update balance
    - balance is 1150 at this point
  - T1 - update balance
    - balance is 950 at this point
  - **The final value of balance is 950**

Thread synchronization

- **Mutex (mutual exclusion)**
  - Thread 1
    - Mutex.lock()
      - Read balance
      - Deduct 50 from balance
      - Update balance with new value
    - Mutex.unlock()
  - Thread 2
    - Mutex.lock()
      - Read balance
      - Add 150 to balance
      - Update balance with new value
    - Mutex.unlock()
  - balance = 1100
- Only one thread will get the mutex. Other thread will **block in Mutex.lock()**
- Other thread can start execution only when the thread that holds the mutex calls **Mutex.unlock()**

Lab 6

- Evaluate the performance of multithreaded 'sort' command
  - `od -An -f -N 4000000 < /dev/urandom | tr -s ' ' '\n' > random.txt`
  - Might have to modify the command above
- Delete the empty line
  - `time -p sort -g --parallel=2 numbers.txt > /dev/null`
- Add /usr/local/cs/bin to PATH
  - $ export PATH=/usr/local/cs/bin:$PATH
- Generate a file containing 10M random **double-precision floating point numbers**, one per line with no white space
  - /dev/urandom: pseudo-random number generator

Lab 6

- **`od`**
  - write the contents of its input files to standard output in a user-specified format
  - Options
    - -t f: Double-precision floating point
    - -N <count>: Format no more than *count* bytes of input
- **`sed, tr`**
  - Remove address, delete spaces, add newlines between each float

Lab 6

- use `time -p` to time the command `sort -g` on the data you generated
- Send output to /dev/null
- Run `sort` with the `--parallel` option and the `-g` option: compare by general numeric value
  - Use `time` command to record the real, user and system time when running sort with 1, 2, 4, and 8 threads
    - $ `time -p sort -g file_name > /dev/null` (1 thread)
    - $ `time -p sort -g --parallel=[2, 4, or 8] file_name > /dev/null`
  - Record the times and steps in log.txt

Ray-Tracing

- **Powerful rendering technique in Computer Graphics**
- **Yields high quality rendering**
  - Suited for scences with complex light interactions
  - Visually realistic for a wider variety of materials
  - Trace the path of light in the scene
- **Computationally expensive**
  - Not suited for real-time rendering (e.g. games)
  - Suited for rendering high quality pictures (e.g. movies)
- **Embarrassingly parallel**
  - Good candidate for **multi-threading**
  - Threads need **not synchronize** with each other, because each thread works on a different pixel (at least at small scale)

SSH - Secure Shell

CS 35L
Spring 2018 - Lab 3

Simple Cryptography Crash Course

Communication Over the Internet

- What type of guarantees do we want?
  - **Confidentiality**
    - Message secrecy
  - **Data integrity**
    - Message consistency
  - **Authentication**
    - Identity confirmation
  - **Authorization**
    - Specifying access rights to resources

Cryptography

- Plaintext: actual message
- Ciphertext: encrypted message (unreadable to unintended recipients)
- Encryption: converting from plaintext to ciphertext
- Decryption: converting from ciphertext to plaintext
- Secret key
  - Part of the function used to encrypt\decrypt
  - Good key makes it hard to recover plaintext from ciphertext

Symmetric-key Encrption

- Same secret key used for encryption and decryption
- **Example** : Data Encryption Standard (**DES**)
- **Caesar's cipher**
  - Map the alphabet to a shifted version
    - ABCDEFGHIJKLMNOPQRSTUVWXYZ
    - DEFGHIJKLMNOPQRSTUVWXYZABC
  - Plaintext – SECRET.
  - Ciphertext – VHFUHW Key is 3 (number of shifts of the alphabet)
- **Key distribution** is a problem
  - The secret key has to be delivered in a safe way to the recipient
  - Chance of key being compromised

Image Source: wikipedia

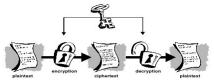


Image Source: gpgtools.org

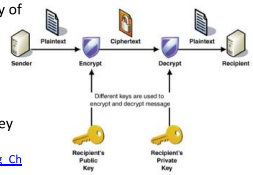
Public-Key Encryption (Asymmetric)

- Uses a pair of keys for encryption
  - Public Key- published and well known to everyone
  - Private- secret key known only to the owner
- Encryption
  - Use public key to encrypt messages
  - Anyone can encrypt message, but they cannot decrypt the ciphertext
- Decryption
  - Use private key to decrypt messages
- In what scheme is this encryption useful?

Public-Key Encryption (Asymmetric)

- Example: RSA (Rivest, Shamir & Adelman)
  - Property used: Difficulty of factoring large integers to prime numbers
  - N = p * q
  - M is a large integer.
  - p,q are prime numbers
  - N is part of the public key

en.wikipedia.org/wiki/RSA_Factoring_Challenge



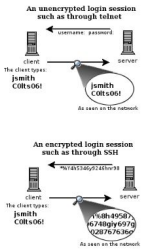
Encryption Types Comparison

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

SSH

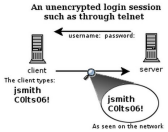
Secure Shell (SSH)

- Telnet
  - Remote access
  - Not encrypted
  - Packet sniffers can intercept sensitive information (username/password)
- SSH
  - run processes remotely
  - encrypted session
  - Session key (secret key) used for encryption during the session



What is SSH?

- **S**ecure **Sh**ell
- Used to remotely access shell
- Successor of telnet
- Encrypted and better authenticated session



CONFIDENTIAL

SSH Initialization

High level description:

1. Negotiating the version of the protocol to use
2. Negotiating cryptographic algorithms, etc.
3. Negotiating a one-time session key for encrypting the rest of the session
4. Authenticating the server host using its host key
5. Authenticating the client using a password, public key authentication, or other means

Session Encryption

- Client and server agree on a **symmetric encryption key** (session key)
- All messages sent between client and server
  - encrypted at the sender with session key
  - decrypted at the receiver with session key
- anybody who doesn't know the session key (hopefully, no one but client and server) doesn't know any of the contents of those messages

High-Level SSH Protocol

- Client ssh's to remote server
  - `$ ssh username@somehost`
  - If first time talking to server -> host validation

```
The authenticity of host 'somehost (192.168.1.1)' can't be established.
RSA key fingerprint is 90:9c:46:ab:03:1d:30:2c:5c:87:c5:c7:d9:13:5d:75.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'somehost' (RSA) to the list of known hosts.
```

  - ssh doesn't know about this host yet
  - shows hostname, IP address and fingerprint of the server's public key, so you can be sure you're talking to the correct computer
  - After accepting, public key is saved in ~/.ssh/known_hosts

Secure Shell (SSH) - Client Authentication

- **Password** login
  - `ssh username@ugrad.seas.ucla.edu`
  - `Enter password`
- **Passwordless** login with keys
  - Use private/public keys for authentication (server and client authentication)
  - ssh-keygen
    - Passphrase (longer version of a password/more secure)
    - Passphrase for protecting the private key
    - Passphrase needed whenever the keys are accessed
  - ssh-copy-id username@ugrad.seas.ucla.edu
    - Copies the public key to the server (~/.ssh/authorized_keys)
  - Login without password
    - ssh username@ugrad.seas.ucla.edu
    - Run scripts/commands on the remote machine
      - ssh username@ugrad.seas.ucla.edu ls
    - But you need to provide a passphrase to use a private key

Secure Shell (SSH) - Client Authentication

- **Passphrase-less** authentication
  - `ssh-agent` → authentication agent
  - Manages private key identities for SSH
  - To avoid entering the passphrase whenever the key is used
  - `ssh-add`
    - Registers the private key with the agent
    - Passphrase asked only once
    - `ssh` will ask the `ssh-agent` whenever the private keys are needed

Parallelization

- **Parallelization** is the practice of accelerating a program by running multiple sections simultaneously
- **Process forking** allows for a process to split into multiple subprocesses that run simultaneously
  - Switching between processes (context switching) on the CPU is expensive
  - Inter-process signalling is difficult (eg. pipes)
- **Multithreading** is an efficient type of parallelization
  - **Thread switches are less expensive**
  - Inter-thread signalling is easy via **shared data**
  - Need **synchronization** among threads accessing the same data
    - e.g. Mutex.lock(), Mutex.unlock()

Pthread API

```
#include <pthread.h>
```

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void* (*thread_function) (void*), void *arg);`
  - Returns 0 on success, otherwise returns non-zero number
- `void pthread_exit(void *retval);`
- `int pthread_join(pthread_t thread, void **retval);`
  - `thread`: thread ID of thread to wait on
  - `retval`: the exit status of the target thread is stored in the location pointed to by `*retval`
    - Pass in NULL if no status is needed
  - Returns 0 on success, otherwise returns non zero error number

Simple Example

```
#include <pthread.h> …
#define NUM_THREADS 5

void *PrintHello(void *thread_ num) {
    printf("\n%d: Hello World!\n", (int) thread_num);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int ret, t;
    for(t = 0; t < NUM_THREADS; t++) {
        ret = pthread_create(&threads[t], NULL,
                             PrintHello, (void*)t);
        // check return value
    }
    for(t = 0; t < NUM_THREADS; t++) {
        ret =  pthread_join(threads[t], NULL);
        // check return value
    }
}
```

Race Conditions

Execution order of threads is non-deterministic

Race Condition:

`Total = Total + val1`	`Total = Total - val2`

What value does Total end with?

Solution: Mutexes for synchronization

Mutex Example (w/o mutexes)

```
#include <pthread.h> …

const int nthreads = 5;
pthread_t tid[nthreads];

int counter;

void* doSomeThing(void *arg) {

    counter = counter + (int)arg;

}

int main() {
    int i;
     counter = 0;

     for (i = 1; i <= nthreads; ++i)
        pthread_create(&(tid[i]), NULL, &doSomeThing, i);
     for (i = 1; i <= nthreads; ++i)
        pthread_join(tid[i], NULL);

     printf("Counter: %d\n", counter);
     return 0;
}
```

Mutex Example (w/ mutexes)

```
#include <pthread.h> …

const int nthreads = 5;
pthread_t tid[nthreads];
pthread_mutex_t lock;
int counter;

void* doSomeThing(void *arg) {
    pthread_mutex_lock(&lock);
    counter = counter + (int)arg;
    pthread_mutex_unlock(&lock);
}

int main() {
    int i;
     counter = 0;
    pthread_mutex_init(&lock, NULL);
     for (i = 1; i <= nthreads; ++i)
        pthread_create(&(tid[i]), NULL, &doSomeThing, i);
     for (i = 1; i <= nthreads; ++i)
        pthread_join(tid[i], NULL);
    pthread_mutex_destroy(&lock);
     printf("Counter: %d\n", counter);
     return 0;
}
```

Deadlock

Deadlock:

`mutex1.lock();`	`mutex2.lock();`
`mutex2.lock();`	`mutex1.lock();`

What happens if each thread is waiting on a resource that is locked by another?

Solutions

- Ignore (simple to implement, but unsafe)
- Detect (slightly complicated): directed graph cycle checking
- Prevent (very complicated): wait-for-graphs, banker's algorithm, etc.

SIMD vs MIMD

- Multiple Instruction Multiple Data (MIMD)
  - Performs multiple actions on any number of data pieces simultaneously.
  - Standard CPU multithreading (eg. pthread)
- Single Instruction Multiple Data (SIMD)
  - Performs the same action on multiple pieces of data simultaneously.
  - Best for algorithms with little data interaction.
  - Typical of most modern parallel specialized hardware, including GPUs (CUDA).

Homework 6

- Download the single-threaded raytracer implementation
- Run it to get output image
- Multithread ray tracing
  - Modify main.c and Makefile
- Run the multithreaded version and compare resulting image with single-threaded one

Homework 6

- Build a multi-threaded version of Ray tracer
- Modify "main.c" & "Makefile"
  - Include <pthread.h> in "main.c"
  - Use "pthread_create" & "pthread_join" in "main.c"
  - Link with –lpthread flag (LDLIBS target)
- make clean check
  - Outputs "1-test.ppm"
  - Can see "1-test.ppm"
  - See next slide on how to convert ppm

Ray-tracing

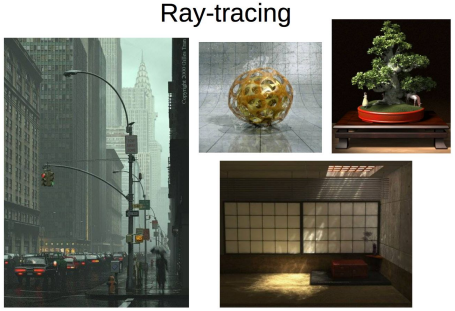


Image Source: POV Ray, Hall of Fame hof.povray.org

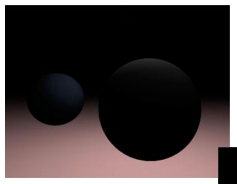
Motivation

Siggraph 2017 technical papers

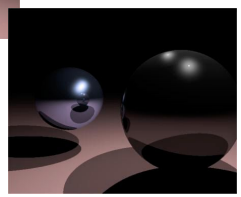
Siggraph Asia 2017 technical papers

Ray-Tracing

- **Powerful rendering technique in Computer Graphics**
- **Yields high quality rendering**
  - Suited for scenes with complex light interactions
  - Visually realistic
  - Trace the path of light in the scene
- **Computationally expensive**
  - Not suited for real-time rendering (e.g. games)
  - Suited for rendering high quality pictures (e.g. movies)
- **Embarrassingly parallel**
  - Good candidate for **multi-threading**
  - Threads need **not synchronize** with each other, because each thread works on a different pixel



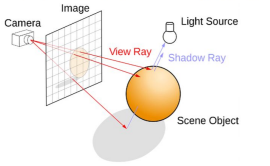
Without ray tracing



With ray tracing

Ray-tracing

- Trace the path of a ray from the eye
  - **One ray per pixel** in the view window
  - The color of the ray is the color of the corresponding pixel
- Check for **intersection** of ray with scene objects.
- **Lighting**
  - **Flat shading** – The whole object has uniform brightness
  - **Lambertian shading** – Cosine of angle between surface normal and light direction



Viewing a ppm file

- How to view a ppm file?
  - ppmtojpeg
    - ppmtojpeg - is more lightweight than gimp. If you don't already have it, you can download ppmtojpeg as part of the Netpbm package here (windows,linux,mac)
    - This program comes with many Linux distributions as well as with Cygwin for Windows; it is also installed on the SEAS Unix machines.
      - ppmtojpeg input-file.ppm > output-file.jpg
  - Gimp:
    - sudo apt-get install gimp (Ubuntu)
    - www.gimp.org - or install on your computer (windows,linux,mac)
      - scp the file to your local folder to view it
        - » conversion tutorial with gimp
      - X forwarding (lnxsrv)
        - » gimp 1-test.ppm

Digital signature

- An electronic stamp\seal
- Digital signature is extra data attached to the document
  - Can be used to check **tampering**
  - Ensures **integrity** of the documents
  - Receiver received the document that the sender intended
- Message digest
  - **Shorter** version of the document
  - Generated using **hashing** algorithms
  - Even a slight change in the original document will change the message digest with **high probability**

Steps for Generating a Digital Signature

SENDER:

1) Generate a *Message Digest*
   - The message digest is generated using a set of hashing algorithms
   - A message digest is a 'summary' of the message we are going to transmit
   - Even the slightest change in the message produces a different digest
2) Create a Digital Signature
   - The message digest is encrypted using the sender's *private* key. The resulting encrypted message digest is the *digital signature*
3) Attach digital signature to message and send to receiver

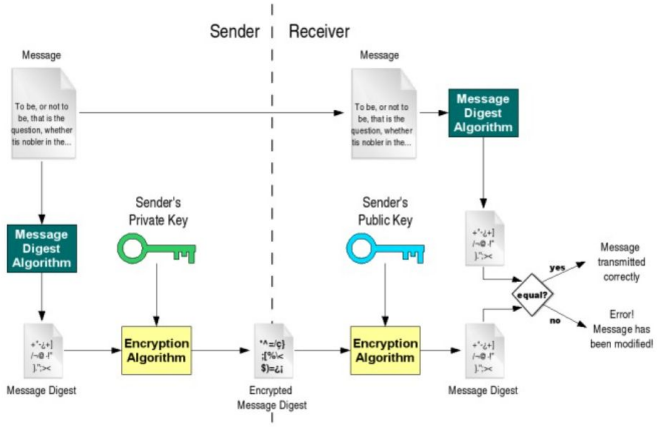
Steps for Generating a Digital Signature

RECEIVER:

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

Digital signature

Verifies document integrity, but does it prove origin? and who is the Certificate Authority?



> gpg [option]

GNU privacy guard

--gen key	generating new keys
--armor	ASCII format
--export	exporting public key
--import	import public key
--detach-sign	creates a file with just the signature
--verify	verify signature with a public key
--encrypt	encrypt document
--decrypt	decrypt document
--list-keys	list all keys in the keyring
--send-keys	register key with a public server/-keyserver option
--search-keys	search for someone's key

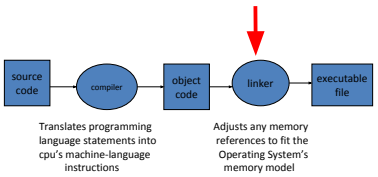
Homework 7

- Answer 2 questions in the file **`hw.txt`**
- Generate a key pair with the GNU Privacy Guard's commands
  - `$ gpg --gen-key` (choose default options)
- Export public key, in ASCII format, into **`hw-pubkey.asc`**
  - `$ gpg --armor --output hw-pubkey.asc --export 'Your Name'`
- Use the private key you created to make a detached clear signature `eeprom.sig` for `eeprom`
  - `$ gpg --armor --output eeprom.sig --detach-sign eeprom`
- Use given commands to verify signature and file formatting
  - These can be found at the end of the assignment spec

Dynamic Linking

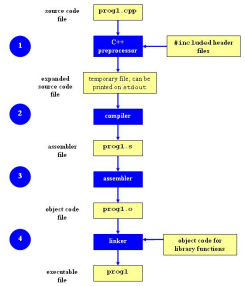
CS 35L
Spring 2018 - Lab 3

Building an executable file



Compilation Process

- Preprocessor
  - Expand Header includes, macros, etc
  - -E option in gcc to show the resulting code
- Compiler
  - Generates machine code for certain architecture
- Linker
  - Linkall modules together
  - Address resolution
- Loader
  - Loads the executable to memory to start execution

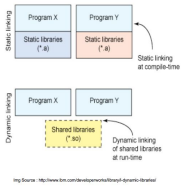


Linking and Loading

- Linker collects procedures and links them together object modules into one executable program
- Why isn't everything written as just one **big** program, saving the necessity of linking?
  - Efficiency: if just one function is changed in a 100K line program, why recompile the whole program? Just recompile the one function and relink.
  - Multiple-language programs
  - Other reasons?

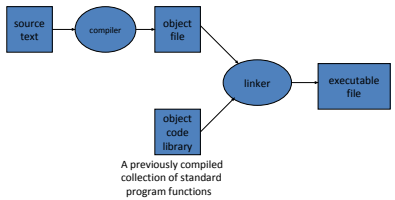
Linux Libraries

- **Static Library**
  - Statically linked
  - Every program has its own copy
  - More space in memory
  - Tied to a specific version of the lib. New version of the lib requires recompile of source code.
- **Shared Library (binding at run-time)**
  - Dynamically loaded/linking
    - **Dynamic Linking** – The OS loads the library when needed. A dynamic linker does the linking for the symbol used.
    - **Dynamic Loading** – The program "actively" loads the library it needs (DL API – dlopen(), dlclose()). More control to the program at run-time. Permits extension of programs to have new functionality.
  - Library is shared by multiple programs
  - Lower memory footprint
  - New version of the lib does not require a recompile of source code using the lib



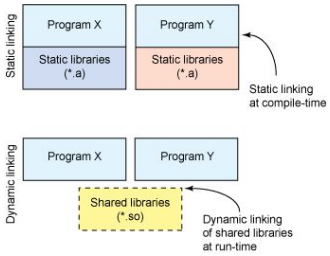
Static Linking

- Carried out only once to produce an executable file
- If static libraries are called, the linker will copy all the modules referenced by the program to the executable
- Static libraries are typically denoted by the .a file extension



Dynamic Linking

- Allows a process to add, remove, replace or relocate object modules during its execution.
- If shared libraries are called:
  - Only copy a little reference information when the executable file is created
  - Complete the linking during loading time or running time
- Dynamic libraries are typically denoted by the .so (shared object) file extension
  - .dll (dynamically linked library) on Windows



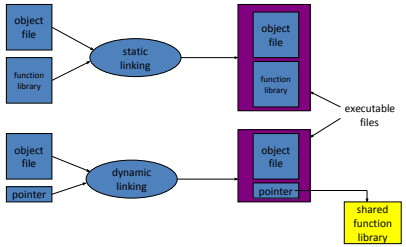
Dynamic linking

- Unix systems: Code is typically compiled as a *dynamic shared object* (DSO)
- Dynamic vs. static linking resulting size

```
$ gcc -static hello.c -o hello-static
$ gcc hello.c -o hello-dynamic
$ ls -l hello
      80 hello.c
   13724 hello-dynamic
     383 hello.s
1688756 hello-static
```

- If you are the sysadmin, which do you prefer?

Smaller is more efficient



Advantages of dynamic linking

- The executable is typically smaller
- When the library is changed, the code that references it does not usually need to be recompiled
- The executable accesses the .so at run time; therefore, multiple programs can access the same .so at the same time
  - Memory footprint amortized across all programs using the same .so

Disadvantages of dynamic linking

- Performance hit
  - Need to load shared objects (at least once)
  - Need to resolve addresses (once or every time)
  - Remember back to the system call assignment…
- What if the necessary dynamic library is missing?
- What if we have the library, but it is the wrong version?

How are libraries dynamically linked?

Table 1. The DI API

Function	Description
dlopen	Makes an object file accessible to a program
dlsym	Obtains the address of a symbol within a `dlopen`ed object file
dlerror	Returns a string error of the last error that occurred
dlclose	Closes an object file

Dynamic loading

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

int main(int argc, char* argv[]) {
    int i = 10;
    void (*myfunc)(int *);
    void *dl_handle;
    char *error;

    dl_handle = dlopen("libmymath.so", RTLD_LAZY); //vs RTLD_NOW
    if (!dl_handle)
        { printf("dlopen() error - %s\n",dlerror()); return 1; }

    myfunc = dlsym(dl_handle, "mul5");
    error = dlerror();
    if (error != NULL)
        { printf("dlsym mul5 error - %s\n", error); return 1; }
    myfunc(&i);

    myfunc = dlsym(dl_handle, "add1");
    error = dlerror();
    if (error != NULL)
        { printf("dlsym add1 error - %s\n", error); return 1; }
    myfunc(&i);

    printf("i = %d\n", i);
    dlclose(dl_handle);
    return 0;
}
```

Copy this code into main.c
`gcc main.c -o main -ldl`

You will have to set the environment variable LD_LIBRARY_PATH to include a path that contai libmymath.so

GCC Flags

- `-fPIC`: Compiler directive to output position independent code, a characteristic required by shared libraries.
- `-lXXX`: Link with "`libXXX.so`"
  - Without `-L` to directly specify the path, `/usr/lib` is used.
- `-L`: At **compile** time, find `.so` from this path.
- `-Wl,rpath=.`: `-Wl` passes options to linker. `-rpath` at **runtime** finds `.so` from this path.
- `-c`: Generate object code from c code.
- `-shared`: Produce a shared object which can then be linked with other objects to form an executable.

Creating static and shared libs in GCC

- mymath.h

```
#ifndef _MY_MATH_H
#define _MY_MATH_H
void mul5(int *i);
void add1(int *i);
#endif
```

- mul5.c

```
#include "mymath.h"
void mul5(int *i)
{
  *i *= 5;
}
```

- add1.c

```
#include "mymath.h"
void add1(int *i)
{
  *i += 1;
}
```

- gcc -c mul5.c -o mul5.o
- gcc -c add1.c -o add1.o
- ar -cvq libmymath.a mul5.o add1.o → (static lib)
- gcc -shared -fpic -o libmymath.so mul5.o add1.o → (shared lib)

Attributes of Functions

- Used to declare certain things about functions called in your program
  - Help the compiler optimize calls and check code
- Also used to control memory placement, code generation options or call/return conventions within the function being annotated
- Introduced by the **attribute** keyword on a declaration, followed by an attribute specification inside double parentheses

Attributes of Functions

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

```
__attribute__ ((__constructor__))
void to_run_before (void) {
   printf("pre_func\n");
}
```

Lab 8

- Write and build simple math program in C
  - compute cos(sqrt(3.0)) and print it using the printf format "%.17g"
  - Use `ldd` to investigate which dynamic libraries your hello world program loads
  - Use `strace` to investigate which system calls your hello world program makes
- Use "`ls /usr/bin | awk 'NR%101==SID%101'`" to find +-23 linux commands to use `ldd` on
  - Record output for each one in your log and investigate any errors you might see
  - From all dynamic libraries you find, create a sorted list
    - Remember to remove the duplicates!

Lab 8 hint

```
#!/bin/bash

for x in "$(ls /usr/bin | awk \
'NR%101==your_uid%101' $1)"; do
  y=`which $x`
  ldd $y
done
```

example run, unique sort, need to omit addresses at end:

```
./ldd_run | grep so | sort -u
```

Homework 8

- Split `randall.c` into 4 separate files
- Stitch the files together via static and dynamic linking to create the program
- `randmain.c` must use *dynamic loading, dynamic linking* to link up with `randlibhw.c` and `randlibsw.c` (using `randlib.h`)
- Write the `randmain.mk` makefile to do the linking

Homework 8

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

Homework 8

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

Useful Resources

- Useful overview of dynamic libraries: Click here
- Man page for DL API: Click here
- www.ibm.com/developerworks/library/l-dynamic-libraries/
- www.ibm.com/developerworks/library/l-lpic1-102-3/
- tldp.org/HOWTO/Program-Library-HOWTO/index.html
- www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html
- man7.org/linux/man-pages/man7/vdso.7.html

Software development process

- Involves making a lot of changes to code
  - New features
  - Bug fixes
  - Performance enhancements
- Many people editing code simultaneously need to:
  - Compare different versions
  - Combine different versions into a new version
  - Reference previous versions
- Multiple versions of dependencies, environments

What Changes Are We Managing?

Software
- Planned software development
  - team members constantly add new code
- (Un)expected problems
  - bug fixes
- Enhancements
  - Make code more efficient (memory, execution time)

"The only constant in software development is change"

Features Required to Manage Change

- Backups
- Timestamps
- Who made the change?
- Where was the change made?
- A way to communicate changes with team

How to achieve that

- Big project with multiple files
  - Bug fix required changing multiple files
  - Bug fix didn't work
  - How to find the problem
  - … Or how to revert to a version before the bug
- Figure out which parts changed (**diff**?)
- Communicate changes with team (**patch**?)
- But diff and patch are not that good

Disadvantages of diff & patch

- Diff requires keeping a copy of old file before changes
- Work with only 2 versions of a file (old & new)
  - Projects will likely be updated more than once
  - store versions of the file to see how it evolved over time

```
index.html
  index-2009-04-08.html
  index-2009-06-06.html
  index-2009-11-04.html
  index-2010-01-23.html
```

- Numbering scheme becomes more complicated if we need to store two versions for the same date

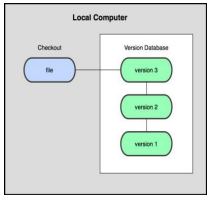
Disadvantages of diff & patch

- Two people may edit the same file on the same date
  - 2 patches need to be sent and merged
- Changes to one file might affect other files (eg. .h & .c)
  - Need to make sure those versions are stored together as a group

Source Control Software (SCS)

- Also called Version Control Software (VCS)
- Track changes to code and other files related to software
  - What new files were added?
  - What changes made to files?
  - Which version had what changes?
  - Which user made the changes?
  - Revert to previous version
- Track entire history of software
- Source control software (SCS)
  - Git, Subversion (SVN), CVS, and others

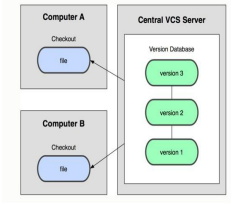
Local SCS



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

Image Source: git-scm.com

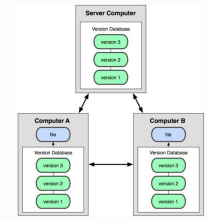
Centralized SCS



- Version history sits on a central server
- Users will get a working copy of the files
- Changes have to be committed to the server
- All users can get the changes

Image Source: git-scm.com

Distributed SCS



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

Image Source: git-scm.com

Terms used

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

Centralized vs. Distributed SCS

- Single central copy of the project history on a server
- Changes are uploaded to the server
- Other programmers can get changes from the server
- Examples: SVN, CVS
- Each developer gets the full history of a project on their own hard drive
- Developers can communicate changes between each other without going through a central server
- Examples: **Git**, Mercurial, Bazaar, Bitkeeper

Centralized: Pros and Cons

"The full project history is only stored in one central place."

Pros
- Everyone can see changes at the same time
- Simple to design

Cons
- Single point of failure (no backups!)
- Communicating changes between users requires physical or P2P connection

Distributed: Pros and Cons

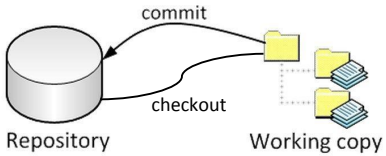
"The entire project history is downloaded to the hard drive"

Pros
- Commit changes/revert to an old version while offline
- Commands run extremely fast because tool accesses the hard drive and not a remote server
- Share changes with a few people before showing changes to everyone

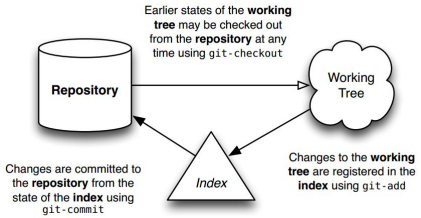
Cons
- Long time to download
- A lot of disk space to store all versions

Git Source Control

Big Picture



Git Workflow



Git commands

- Repository creation
  - **`git init`** (start a new repository)
  - **`git clone`** (create a copy of an existing repository)
- Branching
  - **`git branch <new_branch_name>`** (creates a new branch)
  - **`git checkout <name>`** (switch to a branch or commit with name)
  - **`git checkout -b <new_branch_name>`** (creates and checks out a new branch)
- Commits
  - **`git add`** (stage modified files)
  - **`git commit`** (check-in changes on the current branch)
- Getting info
  - **`git status`** (shows modified files, new files, etc)
  - **`git diff`** (compares working copy with staged files)
  - **`git log`** (shows history of commits)
- Get help with: **`git help`** (or with git's online documentation)

Git Repository Objects

- Objects used by Git to implement source control
  - **Blobs**
    - Sequence of bytes
  - **Trees**
    - Groups blobs/trees together
  - **Commit**
    - Refers to a particular "git commit"
    - Contains all information about the commit
  - **Tags**
    - A named commit object for convenience (e.g. versions of software)
- Objects uniquely identified with **hashes**

Terms used

- **Head**
  - Refers to a commit object
  - There can be many heads in a repository
- **HEAD**
  - Refers to the currently active head
- **Detached HEAD**
  - If a commit is not pointed to by a branch
  - This is okay if you want to just take a look at the code and if you don't commit any new changes
  - If the new commits have to be preserved then a new branch has to be created
    - `git checkout v3.0 -b BranchVersion3.1`
- **Branch**
  - Refers to a head and its entire set of ancestor commits
- **Master**
  - Default branch

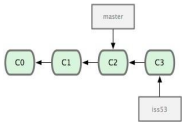


Image Source: git-scm.com

Git States

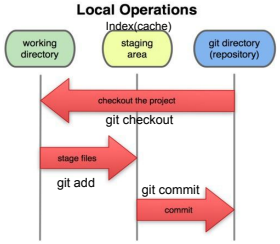


Image Source: git-scm.com

First Git Repository

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

Working With Git

- `$ echo "I love Git" >> hello.txt`
- `$ git status`
  - Shows list of modified files
  - hello.txt
- `$ git diff`
  - Shows changes we made compared to index
- `$ git add hello.txt`
- `$ git diff`
  - No changes shown as diff compares to the index
- `$ git diff HEAD`
  - Now we can see changes in working version
- `$git commit -m 'Second commit'`

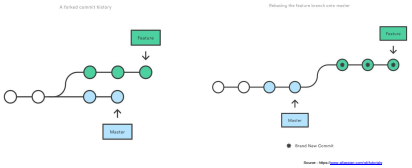
Undoing What Is Done

- **git checkout**
  - Used to checkout a specific version/branch of the tree
  - `git rebase master` (returns to current working version)
- **git revert**
  - Reverts a commit
  - Does not delete the commit object, just applies a patch
  - Reverts can themselves be reverted!
- **Git never deletes a commit object**
  - It is very hard to lose data

Git Rebase

- Rewrites commit history.
- Loses context
- Never use this on public branches!
- How to rebase?

```
$ git checkout feature
$ git rebase master
```



Merging

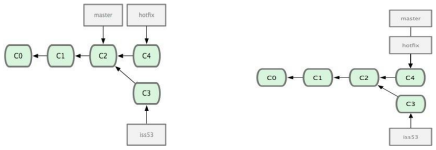


Image Source: git-scm.com

- Merging hotfix branch into master
  - git checkout master
  - git merge hotfix
- Git tries to merge automatically
  - Simple if it is a forward merge
  - Otherwise, you have to manually resolve conflicts

Merging

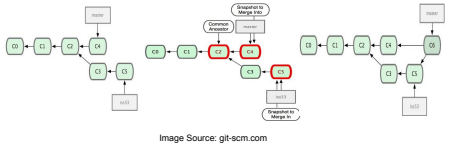


Image Source: git-scm.com

- Merge iss53 into master
- Git tries to merge automatically by looking at the changes since the common ancestor commit
- Manually merge using 3-way merge or 2-way merge
  - Merge conflicts - Same part of the file was changed differently

Merging

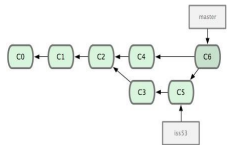


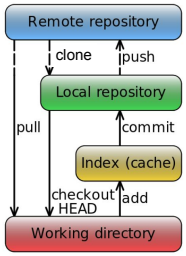
Image Source: git-scm.com

- Refer to multiple parents
  - git show hash
  - git show hash^2 (shows second parent)
- HEAD^^ == HEAD~2

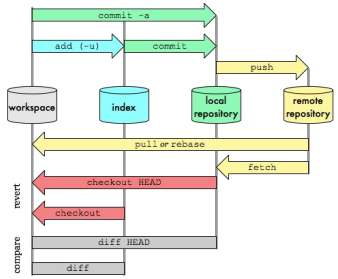
More Git commands

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

Overview



Git Data Transport Commands



Assignment 9

- GNU Diffutils uses " ` " in diagnostics
  - Example: `diff . -`
  - Output: diff: cannot compare – to a directory
  - Want to use apostrophes only
- Diffutils maintainers have a patch for this problem called "maint: quote 'like this' or "like this", not `like this'"
- Problem: You are using Diffutils version 3.0, and the patch is for a newer version

Assignment 1

Unix

GNU/Linux

- Open-source operating system
  - **Kernel**: core of operating system
    - Allocates time and memory to programs
    - Handles file system and communication between software and hardware
  - **Shell**: interface between user and kernel
    - Interprets commands user types in
    - Takes necessary action to cause commands to be carried out
  - **Programs**

Files and Processes

- Everything is either a **process** or a **file**:
  - **Process**: an executing program identified by PID
  - **File**: collection of data
    - A document
    - Text of program written in high-level language
    - Executable
    - Directory
    - Devices

The Basics: Shell

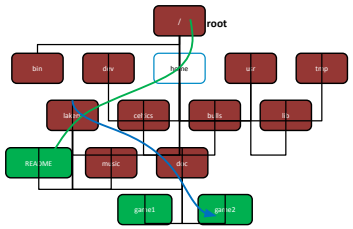
Some of the CLI utilities from you should be familiar with:

- `pwd`
- `cd`
- `mv`
- `cp`
- `rm`
- `mkdir`
- `rmdir`
- `ls`
- `ln`
- `touch`
- `find`
- `which`
- `man`
- `ps`
- `kill`
- `diff`
- `wget`
- `tr`
- `wc`
- `grep`
- `and others...`

The Basics: Shell

- How do I find where files are on the system?
- How do I find out what options are available for a particular utility?
- When is a file a file and when is it a process?
- What types of links are there?

Absolute Path vs. Relative Path



Current directory: home What are the differences between absolute and relative paths?

Linux File Permissions

- chmod
  - read (r), write (w), executable (x)
  - User, group, others
- Why do we have permissions at all?

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

Assignment 2

Shell Scripting

Locale

A locale

- Set of parameters that define a user's cultural preferences
  - Language
  - Country
  - Other area-specific things
- What else does the locale affect?

`locale` command
prints information about the current locale environment to standard output

Environment Variables

- Variables that can be accessed from any child process
- Why do we have these at all? What functions do they serve?

Common ones:

- **HOME**: path to user's home directory
- **PATH**: list of directories to search in for command to execute
- Change value:
  export VARIABLE=…

Locale Settings Can Affect Program Behavior!!

Default sort order for the `sort` command depends:

- LC_COLLATE='C': sorting is in ASCII order
- LC_COLLATE='en_US': sorting is case insensitive except when the two strings are otherwise equal and one has an uppercase letter earlier than the other.

Other locales have other sort orders!

Compiled vs. Interpreted

Compiled languages

- Programs are translated from their original source code into machine code that is executed by hardware
- Efficient and fast
- Require recompiling
- Work at low level, dealing with bytes, integers, floating points, etc.
- Ex: C/C++
- When would I want to use a compiled language?

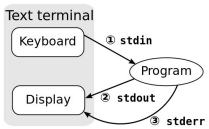
Interpreted languages

- Interpreter program (the shell) reads commands, carries out actions commanded as it goes
- Much slower execution
- Portable
- High-level, easier to learn
- Ex: PHP, Ruby, bash
- When would I want to use an interpreted language?

Why do we have the notion of compiled and interpreted languages?
Why not just have one type of language?

Standard Streams

- Every program has these 3 streams to interact with the world
  - stdin (0): contains data going into a program
  - stdout (1): where a program writes its output data
  - stderr (2): where a program writes its error msgs



Redirection and Pipelines

- *program < file* redirects *file to programs's stdin*:
  `cat <file`
- *program > file* redirects *program*'s stdout to *file2*:
  `cat <file >file2`
- *program 2> file* redirects *program*'s stderr to *file2*:
  `cat <file 2>file2`
- *program >> file* **appends** program's stdout to *file*
- *program1 | program2* assigns stdout of *program1* as the stdin of *program2; text 'flows' through the pipeline*
  `cat <file | sort >file2`

Why would we want to redirect I/O? What are some examples of use cases for I/O redirection? How do we implement this in C?

Regular Expressions

- Notation that lets you search for text with a particular pattern:
  - For example: starts with the letter a, ends with three uppercase letters, etc.
- Why do these exist? Why not just program our own text searching? Are the expressions the same across languages? Platforms?
- What's the difference between a basic and an extended regular expression? When would I use either?
- How do I write a regular expression to accomplish x?

- http://regexpal.com/ to test your regex expressions
- Simple regex tutorial http://www.icewarp.com/support/online_help/203030104.htm

4 Basic Concepts

- Quantification
  - How many times of previous expression?
  - Most common quantifiers: ?(0 or 1), *(0 or more), +(1 or more)
- Grouping
  - Which subset of previous expression?
  - Grouping operator: ()
- Alternation
  - Which choices?
  - Operators: [] and |
    - Hello|World [A B C]
- Anchors
  - Where?
  - Characters: ^ (beginning) and $ (end)
- How do I use a combination of the above to accomplish tasks?

Regular Expressions

Character	BRE / ERE	Meaning in a pattern
\	Both	Usually, turn off the special meaning of the following character. Occasionally, enable a special meaning for the following character, such as for \(...\) and \{...\}.
.	Both	Match any single character except NUL. Individual programs may also disallow matching newline.
*	Both	Match any number (or none) of the single character that immediately precedes it. For EREs, the preceding character can instead be a regular expression. For example, since . (dot) means any character, .* means "match any number of any character." For BREs, * is not special if it's the first character of a regular expression.
^	Both	Match the following regular expression at the beginning of the line or string. BRE: special only at the beginning of a regular expression. ERE: special everywhere.

Regular Expressions (cont'd)

$	Both	Match the preceding regular expression at the end of the line or string. BRE: special only at the end of a regular expression. ERE: special everywhere.
[...]	Both	Termed a bracket expression, this matches any one of the enclosed characters. A hyphen (-) indicates a range of consecutive characters. (Caution: ranges are locale-sensitive, and thus not portable.) A circumflex (^) as the first character in the brackets reverses the sense: it matches any one character not in the list. A hyphen or close bracket (]) as the first character is treated as a member of the list. All other metacharacters are treated as members of the list (i.e., literally). Bracket expressions may contain collating symbols, equivalence classes, and character classes (described shortly).
\{*n*,*m*\}	BRE	Termed an *interval expression*, this matches a range of occurrences of the single character that immediately precedes it. \{*n*\} matches exactly n occurrences, \{*n*,\} matches at least n occurrences, and \{*n*,*m*\} matches any number of occurrences between n and m. n and m must be between 0 and RE_DUP_MAX (minimum value: 255), inclusive.
\(\)	BRE	Save the pattern enclosed between \(and \) in a special *holding space*. Up to nine subpatterns can be saved on a single pattern. The text matched by the subpatterns can be reused later in the same pattern, by the escape sequences \1 to \9. For example, **\(ab\).*\1** matches two occurrences of ab, with any number of characters in between.

Regular Expressions (cont'd)

\n	BRE	Replay the nth subpattern enclosed in \(and \) into the pattern at this point. n is a number from 1 to 9, with 1 starting on the left.
{*n*,*m*}	ERE	Just like the BRE \{*n*,*m*\} earlier, but without the backslashes in front of the braces.
+	ERE	Match one or more instances of the preceding regular expression.
?	ERE	Match zero or one instances of the preceding regular expression.
|	ERE	Match the regular expression specified before or after.
()	ERE	Apply a match to the enclosed group of regular expressions.

Matching Multiple Characters with One Expression

*	Match zero or more of the preceding character
{*n*}	Exactly n occurrences of the preceding regular expression
{*n*,}	At least n occurrences of the preceding regular expression
{*n*,*m*}	Between n and m occurrences of the preceding regular expression

Examples

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

Text Processing Tools

- You should be familiar with:
  - `wc`: outputs a one-line report of lines, words, and bytes
  - `head`: extract top of files
  - `tail`: extracts bottom of files
  - `sort`: sort lines of text files
  - `comm`: compare multiple files
  - `tr`: translate or delete characters
  - `grep`: print lines matching a pattern
  - `sed`: filtering and transforming text
- What are the differences between tr, sed, and grep?
- When would I use each one?
- How can I combine and use these tools together?

`wc`, `head`, and `tail`

`wc`: print line, word, and byte counts for each file
- Usage: `wc [OPTION]... [FILE]...`
  - **-m**, **--chars:** print the number of characters
  - **-w**, **--words:** print the number of words
  - **-l**, **--lines:** print the number of newlines

`head`: output the first part of files
- Defaults to displaying the first 10 lines of each file
- Usage: `head [OPTION]... [FILE]...`
  - **-n**, **--lines=***[-]K*: print the first K lines instead of the first 10; with the leading '-', print all but the last K lines of each file

`tail`: output the last part of files
- Defaults to displaying the last 10 lines of each file
- Usage: `tail [OPTION]... [FILE]...`
  - **-n**, **--lines=***[-]K*: print the last K lines instead of the last 10; with the leading '-', print all but the first K lines of each file

`sort`, `comm`, and `tr`

`sort`: sorts **lines** of **text** files
- Usage: sort [OPTION]… [FILE]…
  - Passing filename of - will cause sort to read from stdin
  - **u:** unique sort, removes duplicates
  - **r:** reverse sort order
- Sort order depends on locale (C locale: ASCII sorting)

`comm`: compare two **sorted** files **line by line**
- Usage: comm [OPTION]…FILE1 FILE2
  - **1/2/3:** suppresses given column number of output

`tr`: translate **or** delete characters
- Usage: tr [OPTION]…SET1 [SET2]
  - **c:** use the complement of SET1
  - **d:** delete characters in SET1, do not translate

You've implemented a version of `tr`. How did you do that?

`grep` and `sed`

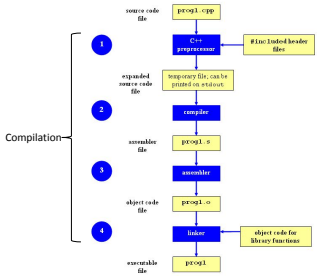
`grep`: print lines matching a pattern
- Usage: `grep [OPTIONS] PATTERN [FILE...]`
  - **-E**: Interpret PATTERN as an extended regular expression (ERE)
    - Defaults to using BRE unless specified otherwise.
  - **-F**: Interpret PATTERN as a list of fixed strings, separated by newlines, any of which is to be matched.
  - **-i**: Ignore case in both the PATTERN and the input files.
  - **-v**: Invert the sense of matching, to select non-matching lines.

`sed`: stream editor for filtering and transforming text
- Usage: `sed [OPTION]... PATTERN [input-file]...`
  - PATTERN must represent: '`s/regExpr/replText/`' where
    - `regExpr` is the pattern to be replaced
    - `replText` is the replacement for text matching `regExpr`
  - **-r**: use extended regular expressions in the pattern

Assignment 3

Modifying and rewriting software

Compilation Process



Compilation Process

- Why do we have this process?
- What are the different components of the process?
  - "I just typed gcc to compile my programs… does that mean gcc has all of the components within it?"
- Why can't I execute individual object code files?
- What are the differences between open source and closed source software? When would I want to use one or the other?

Make

- Utility for managing large software projects
- Compiles files and keeps them up-to-date
- Efficient Compilation (only files that need to be recompiled)
- Why do we have make at all?
  - why don't we just run 'gcc …' from the terminal

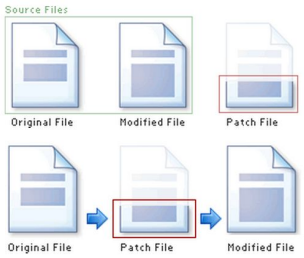
Build Process

- **configure**
  - Script that checks details about the machine before installation
    - Dependency between packages
  - Often creates 'Makefile'
- **make**
  - Requires 'Makefile' to run
  - Compiles all the program code and creates executables in current temporary directory
- **make install**
  - make utility searches for a label named install within the Makefile, and executes only that section of it
  - executables are copied into the final directories (system directories)

Patching

- A patch is a piece of software designed to fix problems with or update a computer program
- It's a diff file that includes the changes made to a file
- A person who has the original (buggy) file can use the patch command with the diff file to add the changes to their original file
- Why not just change the original source code to fix it? Why do we have patches?

Applying a Patch



diff Unified Format

- diff –u original_file modified_file
- --- path/to/original_file
- +++ path/to/modified_file
- @@ -l,s +l,s @@
  - @@: beginning of a 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

What is Python?

- Not just a scripting language
- Object-Oriented language
  - Classes
  - Member functions
- Compiled and interpreted
  - Python code is compiled to bytecode
  - Bytecode interpreted by Python interpreter
- Not as fast as C but easy to learn, read and use
- Why is python powerful? Why is it popular?
- You should know how to write basic python programs

Assignment 4

C Programming and Debugging

C Language

- Subset of C++ (very similar)
- Compiling 'C' only: gcc -std=c99 binsortu.c
- Built-in types:
  - Integers, Floating-point, character strings
  - No bool, false is 0 and true is anything else
- No classes, but we have **structs**
  - No methods and access modifiers

```
struct Song {
    short duration;
    struct Date published;
};
```

Pointers

`double x, y, *ptr;`	Two double variables and a pointer to double.
`ptr = &x;`	Let ptr point to x.
`*ptr = 7.8;`	Assign the value 7.8 to the variable x.
`*ptr *= 2.5;`	Multiply x by 2.5.
`y = *ptr + 0.5;`	Assign y the result of the addition x + 0.5.

Pointers to Functions

```
double (*funcPtr)(double, double);
double result;

// Let funcPtr point to the function pow( ).
// The expression *funcPtr now yields the
// function pow( ).
funcPtr = pow;

// Call the function referenced by funcPtr.
result = (*funcPtr)( 1.5, 2.0 );

// The same function call.
result = funcPtr( 1.5, 2.0 );
```

Dynamic Memory Management

- **malloc(size_t size)**: allocates a block of memory whose size is at least *size*.
- **free(void *ptr)**: frees the block pointed to by ptr
- **realloc(void *ptr, size_t newSize)**: Resizes allocated block

```
Rectangle_t *ptr =
(Rectangle_t*)malloc(sizeof(Rectangle_t));

if(ptr == NULL) {
    printf("Malloc failed!");
    exit(-1);
}
else {
    //Perform tasks with the memory
    free(ptr);
    ptr = NULL;
}

ptr = (Rectangle_t*)
    realloc(ptr, 3*sizeof(Rectangle_t))
```

Opening & Closing Files

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

Common Streams and their file pointers

Standard input: `stdin`

Standard output: `stdout`

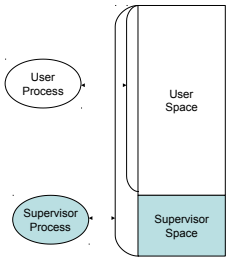
Standard error: `stderr`

Assignment 5

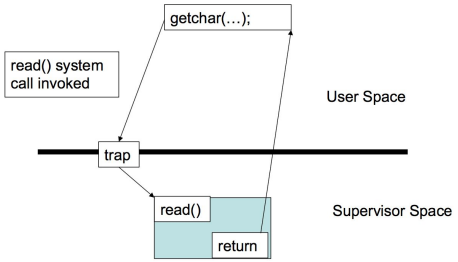
System Call Programming

Processor Modes

- Mode bit may define areas of memory to be used when the processor is in supervisor mode vs user mode



System Calls



Trap: System call causes a switch from user mode to kernel mode

System calls

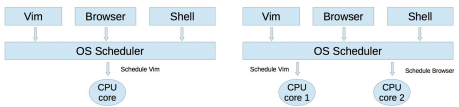
- `ssize_t read(int fildes, void *buf, size_t nbyte)`
  - fildes: file descriptor
  - buf: buffer to write to
  - nbyte: number of bytes to read
- `ssize_t write(int fildes,const void *buf,size_t nbyte)`
  - fildes: file descriptor
  - buf: buffer to write to
  - nbyte: number of bytes to write
- `int open(const char *pathname,int flags,mode_t mode)`
- `int close(int fd)`
- `int fstat(int fd, struct stat *buf)`
  - Returns information about the file with the descriptor fd to buf
- File descriptors:
  - 0 stdin, 1 stdout, 2 stderr
- *Why are these system calls and not just regular library functions?*

Assignment 6

Multithreading

Multitasking

- Run multiple processes **simultaneously** to increase performance
- Processes do not share internal structures (stacks,globals,etc)
  - Communicate via **IPC** (inter-process communication) methods
    - Pipes, sockets, signals, message queues
- **Single core: Illusion** of parallelism by switching processes quickly (**time-sharing**). Why is illusion good?
- **Multi-core: True** parallelism. Multiple processes execute **concurrently** on different CPU cores



Multithreading properties

- Efficient way to **parallelize** tasks
- **Thread switches are less expensive** compared to process switches (context switching)
- Inter-thread communication is easy, via **shared global** data
- Need **synchronization** among threads accessing same data

Pthread API

```
#include <pthread.h>
```

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void* (*thread_function) (void*), void *arg);`
  - Returns 0 on success, otherwise returns non-zero number
- `void pthread_exit(void *retval);`
- `int pthread_join(pthread_t thread, void **retval);`
  - Returns 0 on success, otherwise returns non zero error number

Thread synchronization

- **Mutex (mutual exclusion)**
  - Thread 1
    - Mutex.lock()
      - Read balance
      - Deduct 50 from balance
      - Update balance with new value
    - Mutex.unlock()
  - Thread 2
    - Mutex.lock()
      - Read balance
      - Add 150 to balance
      - Update balance with new value
    - Mutex.unlock()
  - balance = 1100
- Only one thread will get the mutex. Other thread will **block in Mutex.lock()**
- Other thread can start execution only when the thread that holds the mutex calls **Mutex.unlock()**

Questions

- What can go wrong in multithreading?
  - What are race conditions?
  - What is deadlock?
- What are some approaches to make multithreading safer?
  - What are the possible advantages or disadvantages of each of these approaches?
- Comment: It may be useful to consider changing an algorithm to make it safer . . .

Assignment 7

SSH

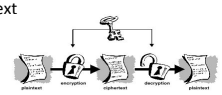
Communication Over the Internet

What type of guarantees do we want?

- **Confidentiality**
  - Message secrecy: "Can anybody else read this message?"
- **Data integrity**
  - Message consistency: "Has someone altered this message?"
- **Authentication**
  - Identity confirmation: "Is this message really from Alice?"
- **Authorization**
  - Predicated permission: "Can bad people get into my system?"

Cryptography

- Plaintext - actual message
- Ciphertext - encrypted message (unreadable gibberish)
- Encryption - converting from plaintext to ciphertext
- Decryption - converting from ciphertext to plaintext
- Secret key
  - part of the mathematical function used to encrypt\decrypt
  - Good key makes it hard to get back plaintext from ciphertext



Symmetric-key Encrption

- Same secret key used for encryption and decryption
- **Example** : Data Encryption Standard (**DES**)
- **Caesar's cipher**
  - Map the alphabet to a shifted version
    - ABCDEFGHIJKLMNOPQRSTUVWXYZ
    - DEFGHIJKLMNOPQRSTUVWXYZABC
  - Plaintext – SECRET. Ciphertext – VHFUHW
  - Key is 3 (number of shifts of the alphabet)
- **Key distribution** is a problem
  - The secret key has to be delivered in a safe way to the recipient
  - Chance of key being compromised

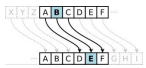


Image Source: wikipedia

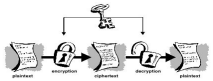


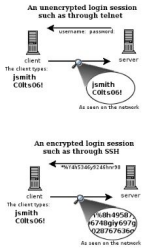
Image Source: gogbooks.org

Public-Key Encryption (Asymmetric)

- Uses a pair of keys for encryption
  - Public Key - published and well known to everyone
  - Private - Secret key known only to the owner
- Encryption
  - Use public key to encrypt messages
  - Anyone can encrypt message, but they cannot decrypt the ciphertext
- Decryption
  - Use private key to decrypt messages
- In what scheme is this encryption useful?

Secure Shell (SSH)

- Telnet
  - Remote access
  - Not encrypted
  - Packet sniffers can intercept sensitive information (username/password)
- SSH
  - run processes remotely
  - encrypted session
  - Session key (secret key) used for encryption during the session



Session Encryption

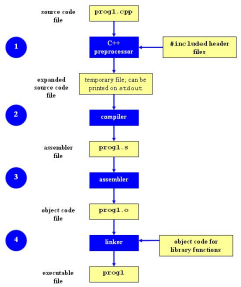
- Client and server agree on a **symmetric encryption key** (session key)
- All messages sent between client and server
  - encrypted at the sender with session key
  - decrypted at the receiver with session key
- anybody who doesn't know the session key (hopefully, no one but client and server) doesn't know any of the contents of those messages

Assignment 8

Dynamic Linking

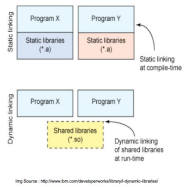
Compilation Process

- Preprocessor
  - Expand Header includes, macros, etc
  - -E option in gcc to show the resulting code
- Compiler
  - Generates machine code for certain architecture
- Linker
  - Linkall modules together
  - Address resolution
- Loader
  - Loads the executable to memory to start execution



Linux Libraries

- **Static Library**
  - Statically linked
  - Every program has its own copy
  - More space in memory
  - Tied to a specific version of the lib. New version of the lib requires recompile of source code.
- **Shared Library (binding at run-time)**
  - Dynamically loaded/linking
    - **Dynamic Linking** – The OS loads the library when needed. A dynamic linker does the linking for the symbol used.
    - **Dynamic Loading** – The program "actively" loads the library it needs (DL API – dlopen(), dlclose()). More control to the program at run-time. Permits extension of programs to have new functionality.
  - Library is shared by multiple programs
  - Lower memory footprint
  - New version of the lib does not require a recompile of source code using the lib



Dynamic Linking

- Allows a process to add, remove, replace or relocate object modules during its execution.
- If shared libraries are called:
  - Only copy a little reference information when the executable file is created
  - Complete the linking during loading time or running time
- Dynamic libraries are typically denoted by the .so (shared object) file extension
  - .dll (dynamically linked library) on Windows

How are libraries dynamically linked?

Table 1. The DI API

Function	Description
dlopen	Makes an object file accessible to a program
dlsym	Obtains the address of a symbol within a `dlopen`ed object file
dlerror	Returns a string error of the last error that occurred
dlclose	Closes an object file

Questions

What are the advantages and disadvantages of dynamic linking (as opposed to static linking)?

How does the build process change with dynamic linking?

What, if any, computational or data overhead could dynamic linking require?

Assignment 9

Change Management

Source/Version Control

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

This seems complicated. Why bother with source control?
What are the strengths and weaknesses of source control?
When would I want to use it? How do I use it?

Terms Used

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

Terms used

- **Head**
  - Refers to a commit object
  - There can be many heads in a repository
- **HEAD**
  - Refers to the currently active head
- **Detached HEAD**
  - If a commit is not pointed to by a branch
  - This is okay if you want to just take a look at the code and if you don't commit any new changes
  - If the new commits have to be preserved then a new branch has to be created
    - `git checkout v3.0 -b BranchVersion3.1`
- **Branch**
  - Refers to a head and its entire set of ancestor commits
- **Master**
  - Default branch

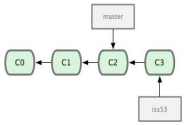
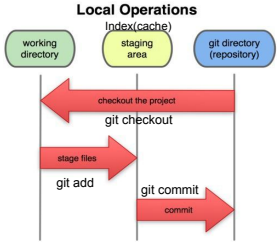


Image Source: git-scm.com

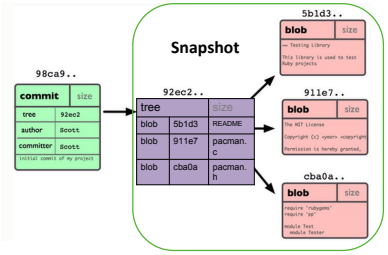
What Is a Branch?

- A pointer to one of the commits in the repo (head) + all ancestor commits
- When you first create a repo, are there any branches?
  - Default branch named 'master'
- The default master branch
  - points to last commit made
  - moves forward automatically, every time you commit

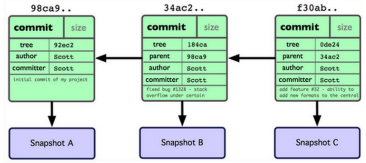
Git States



Git Repo Structure



After 2 More Commits…



Git commands

- Repository creation
  - $ git init (Start a new repository)
  - $ git clone (Create a copy of an exisiting repository)
- Branching
  - $ git checkout <tag/commit> -b <new_branch_name> (creates a new branch)
- Commits
  - $ git add (Stage modified/new files)
  - $ git commit (check-in the changes to the repository)
- Getting info
  - $ git status (Shows modified files, new files, etc)
  - $ git diff (compares working copy with staged files)
  - $ git log (Shows history of commits)
  - $ git show (Show a certain object in the repository)
- Getting help
  - $ git help

You should be familiar with how these commands work and when to use them.

More Git Commands

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

You should be familiar with how these commands work and when to use them.

Questions

- What is the difference between a working copy and the repository?
- What is a commit? What should be in a commit? How many files should commits contain?
- Why bother having branches at all? Why can't we just all work on the same single master branch?
- What happens when we perform a merge? How does it work?