Week 1

1

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**

2

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

3

The Basics: Shell

CLI utilities from week 1 you should be familiar with:

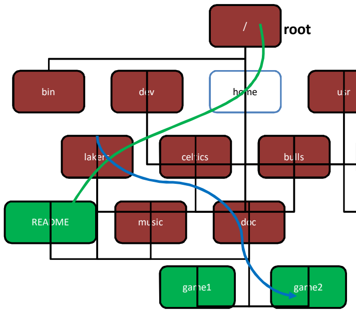
- `pwd`
- `cd`
- `mv`
- `cp`
- `rm`
- `mkdir`
- `rmdir`
- `ls`
- `ln`
- `touch`
- `find`
- `whatis`
- `whereis`
- `man`
- `ps`
- `kill`
- `diff`
- `wget`

4

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

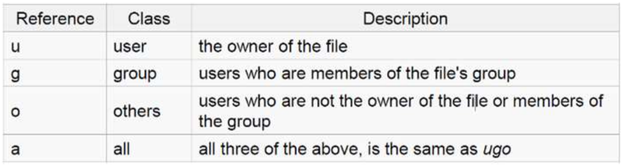


Current directory: home

What are the differences between absolute and relative paths?

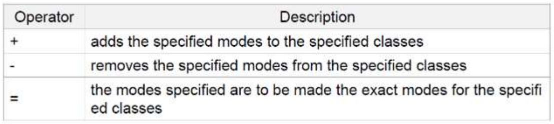
Linux File Permissions

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

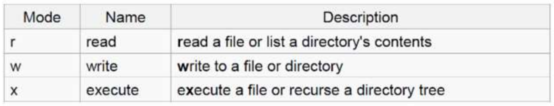


Reference	Class	Description
u	user	the owner of the file
g	group	users who are members of the file's group
o	others	users who are not the owner of the file or members of the group
a	all	all three of the above, is the same as *ugo*

The Basics: chmod (symbolic)

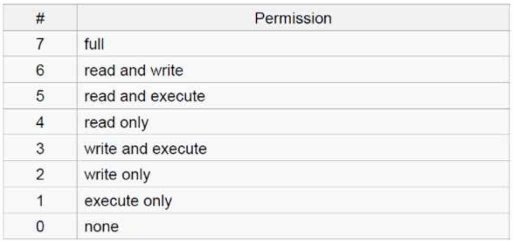


Operator	Description
+	adds the specified modes to the specified classes
-	removes the specified modes from the specified classes
=	the modes specified are to be made the exact modes for the specified classes



Mode	Name	Description
r	read	read a file or list a directory's contents
w	write	write to a file or directory
x	execute	execute a file or recurse a directory tree

The Basics: chmod (numeric)



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

- Usage

– chmod [''references''][''operator''][''modes''] ''file1'' ...
Example: **chmod** ug+rw mydir, **chmod** a-w myfile,
Example: **chmod** ug=rx mydir, **chmod** 664 myfile

9

Week 2/3

10

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

11

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!

12

Environment Variables

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

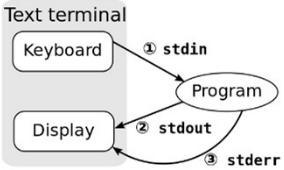
Common ones:

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

13

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



14

Redirection and Pipelines

- *program < file* redirects *file to program'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?

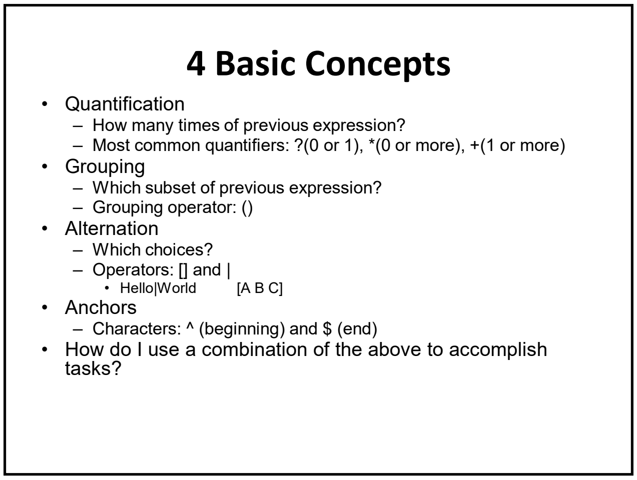
15

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? Are the expressions the same across languages?
- 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

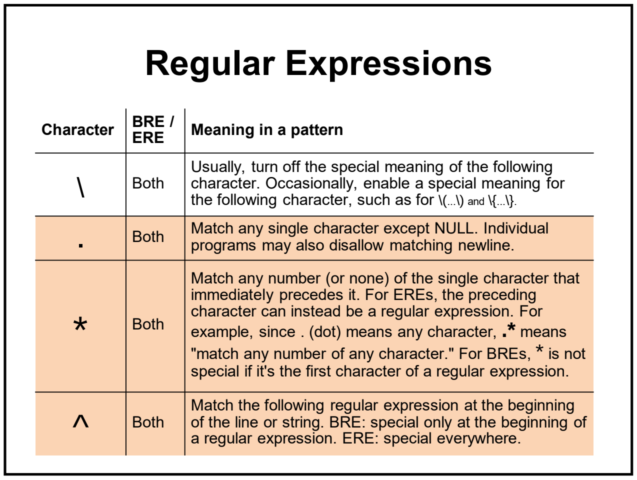
16



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
  - Characters: ^ (beginning) and $ (end)
- How do I use a combination of the above to accomplish tasks?

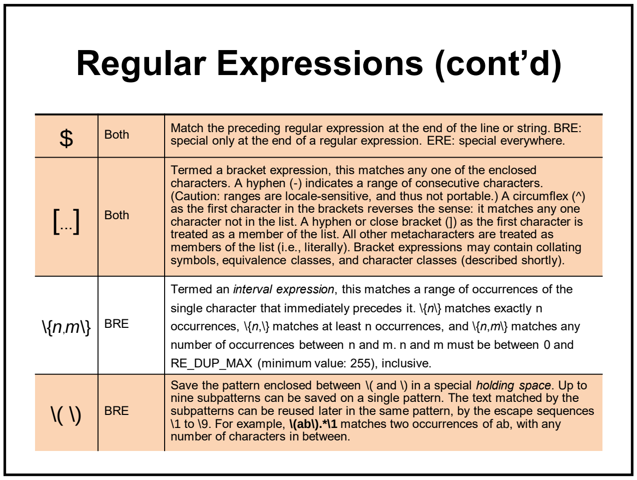
17



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 NULL. 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.

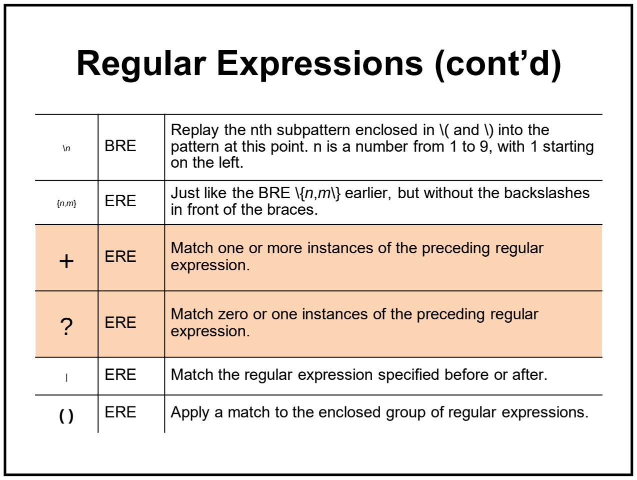
18



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.

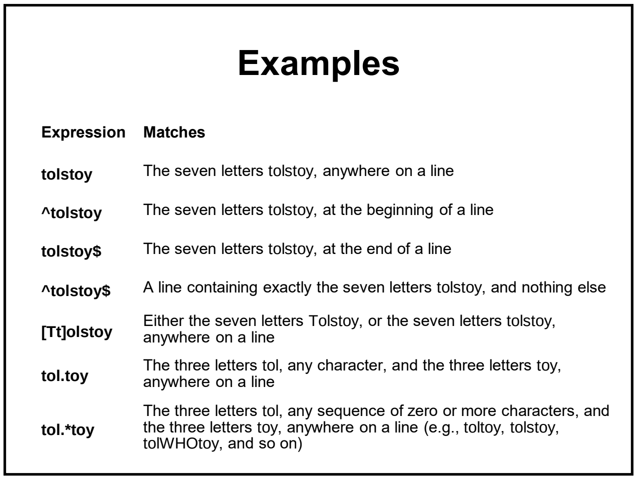
19



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.

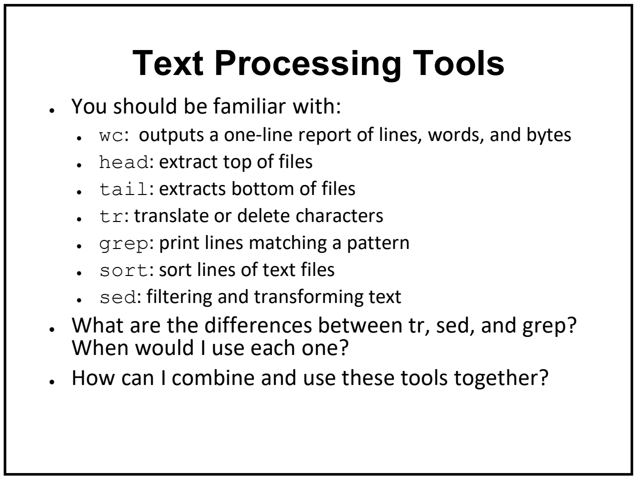
20



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)

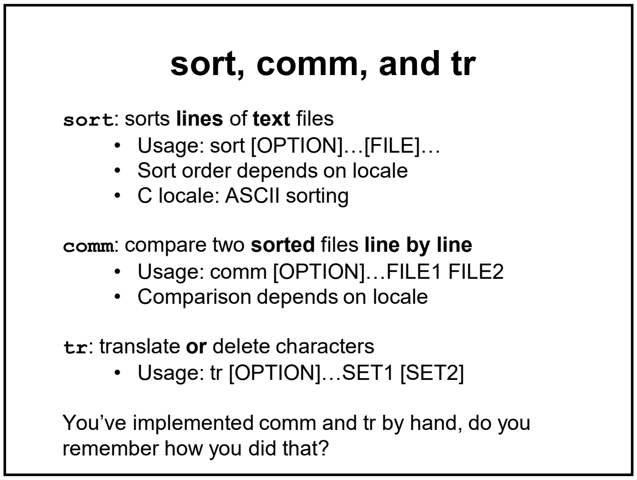
21



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
  - `tr`: translate or delete characters
  - `grep`: print lines matching a pattern
  - `sort`: sort lines of text files
  - `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?

22



sort, comm, and tr

`sort`: sorts **lines** of **text** files

- Usage: sort [OPTION]…[FILE]…
- Sort order depends on locale
- C locale: ASCII sorting

`comm`: compare two **sorted** files **line by line**

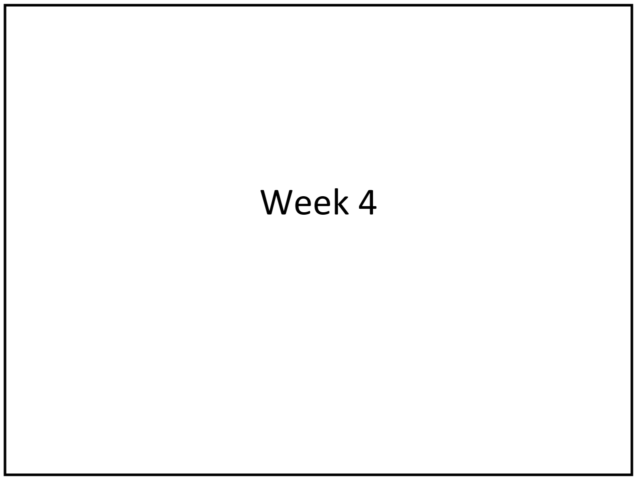
- Usage: comm [OPTION]…FILE1 FILE2
- Comparison depends on locale

`tr`: translate **or** delete characters

- Usage: tr [OPTION]…SET1 [SET2]

You've implemented comm and tr by hand, do you remember how you did that?

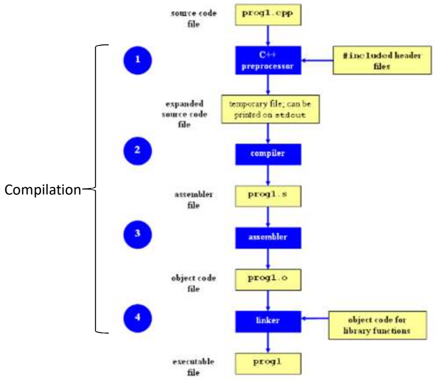
23



Week 4

24

Compilation Process



25

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?

26

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?

27

Build Process

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

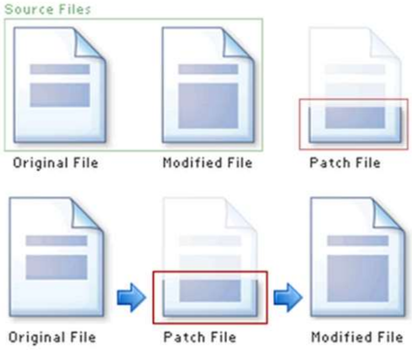
28

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?

29

Applying a Patch



30

diff Unified Format

- diff –u original_file modified_file

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

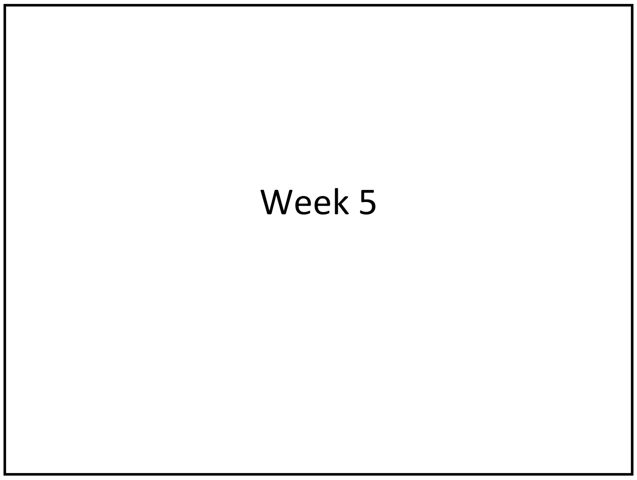
- @@ -l,s +l,s @@
  - @@: beginning of a hunk
  - l: beginning line number
  - s: number of lines the change hunk applies to for each file
  - A line with a:
    - - sign was deleted from the original
    - + sign was added to the original
    - stayed the same

31

What is Python?

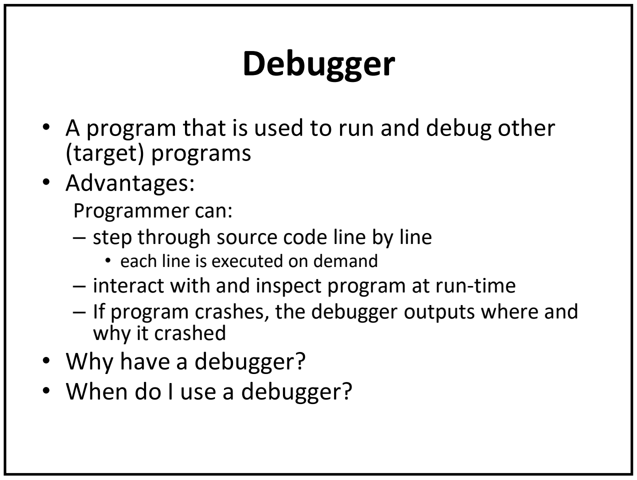
- Not just a scripting language
- Object-Oriented language
  - Classes
  - Member functions
- Not as fast as C but easy to learn, read and use
- You should know how to write basic programs in python

32



Week 5

33

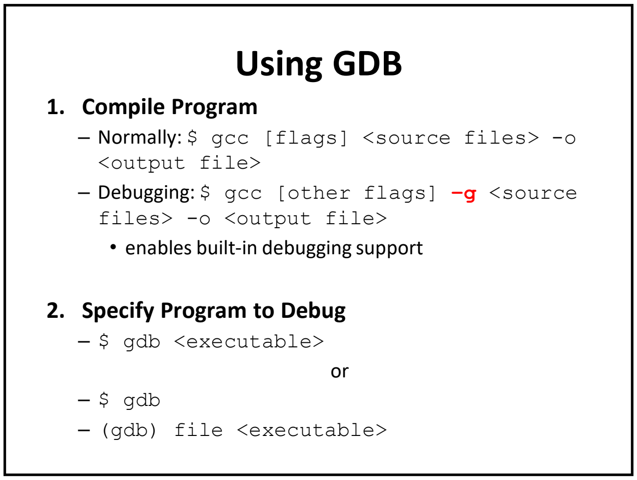


Debugger

- 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 why it crashed
- Why have a debugger?
- When do I use a debugger?

34



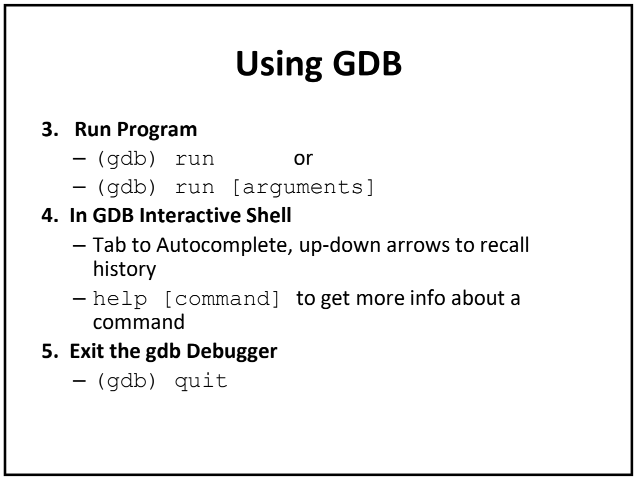
Using GDB

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>`

35



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`

36

Run-Time Errors

- Segmentation fault
  - Program received signal SIGSEGV, Segmentation fault. 0x0000000000400524 in *function* (arr=0x7fffc902a270, r1=2, c1=5, r2=4, c2=6) at *file.c*:12
    - Line number where it crashed and parameters to the function that caused the error
- Logic Error
  - Program will run and exit successfully
- How do we find bugs?

37

Setting Breakpoints

- Breakpoints
  - used to stop the running program at a specific point
  - If the program reaches that location when running, it will pause and prompt you for another command
- Example:
  - `(gdb) break file1.c:6`
    - Program will pause when it reaches line 6 of file1.c
  - `(gdb) break my_function`
    - Program will pause at the first line of `my_function` every time it is called
  - `(gdb) break [`*`position`*`] if `*`expression`*
    - Program will pause at specified position only when the expression evaluates to true
- How do we know where to set breakpoints?
- What do we do once we've stopped at a breakpoint?

38

Deleting, Disabling and Ignoring BPs

- `(gdb) delete [bp_number | range]`
  - Deletes the specified breakpoint or range of breakpoints
- `(gdb) disable [ `*`bp_number`*` | `*`range`*`]`
  - Temporarily deactivates a breakpoint or a range of breakpoints
- `(gdb) enable [ `*`bp_number`*` | `*`range`*`]`
  - Restores disabled breakpoints
- If no arguments are provided to the above commands, all breakpoints are affected!!
- `(gdb) ignore `*`bp_number iterations`*
  - Instructs GDB to pass over a breakpoint without stopping a certain number of times.
    - bp_number: the number of a breakpoint
    - Iterations: the number of times you want it to be passed over

39

Displaying Data

- Why would we want to interrupt execution?
  - to see data of interest at run-time:
  - `(gdb) print [/format] `*`expression`*
    - Prints the value of the specified expression in the specified format
  - Formats:
    - d: Decimal notation (default format for integers)
    - x: Hexadecimal notation
    - o: Octal notation
    - t: Binary notation
- What's the point of displaying data?
- What sort of data might we want to display?

40

Resuming Execution After a Break

- When a program stops at a breakpoint
  - 4 possible kinds of gdb operations:
    - **c or continue**: debugger will continue executing until next breakpoint
    - **s or step**: debugger will continue to next source line
    - **n or next**: debugger will continue to next source line in the current (innermost) stack frame

What is the difference between ‘s’ and ‘n’?
When would we use each one of the above?

41

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**
- Why does the stack exist at all? How is the stack different than the heap?

42

Dynamic Memory

- Memory that is allocated at runtime
  - Why?
- Allocated on the heap
  - Why not the stack?

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*

- What happens if I never call free?
- What happens if I try to put data into dynamic memory but I haven’t yet called malloc?

43

Week 6

44

Processor Modes

- Operating modes that place restrictions on the type of operations that can be performed by running processes
  - User mode: restricted access to system resources
  - Kernel/Supervisor mode: unrestricted access
- System resources?
  - Memory
  - I/O Devices
  - CPU
- Why have different modes?

45

User Mode vs. Kernel Mode

- Hardware contains a mode-bit, e.g. 0 means kernel mode, 1 means user mode
- User mode
  - CPU **restricted** to unprivileged instructions and a specified area of memory
- Supervisor/kernel mode
  - CPU is **unrestricted**, can use all instructions, access all areas of memory and take over the CPU anytime
- What happens if user code is given unrestricted access to CPU?

46

Why Dual-Mode Operation?

- System resources are shared among processes
- OS must ensure:
  - **Protection**
    - an incorrect/malicious program cannot cause damage to other processes or the system as a whole
  - **Fairness**
    - Make sure processes have a fair use of devices and the CPU

47

How to Achieve Protection and Fairness

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

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

48

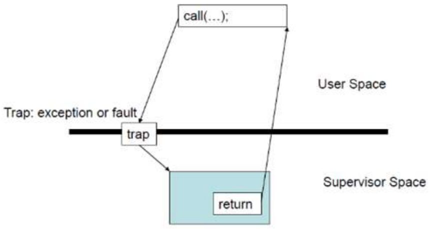
System Calls

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

49

System Calls

- When a system call is made, the program being executed is interrupted and control is passed to the kernel
- If operation is valid the kernel performs it



50

System Call Overhead

- System calls are expensive and can hurt performance
- The system must do many things
  - Process is interrupted & computer saves its state
  - OS takes control of CPU & verifies validity of op.
  - **OS performs requested action**
  - OS restores saved context, switches to user mode
  - OS gives control of the CPU back to user process

51

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
- What are the benefits and tradeoffs of using either system calls or C library functions?

52

Unbuffered vs. Buffered I/O

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

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

Which is faster in what applications? When would you use buffered or unbuffered I/O?

53

Week 7

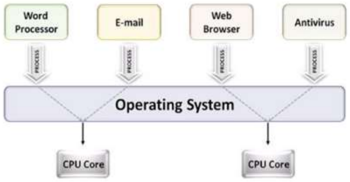
54

Parallelism

- Executing several computations simultaneously to gain performance
- Different forms of parallelism
  - **Multitasking**
    - Several processes are scheduled alternately or possibly simultaneously on a multiprocessing system
  - **Multithreading**
    - Same job is broken logically into pieces (threads) which may be executed simultaneously on a multiprocessing system
- What's the point of parallelism? Isn't it just too complicated?
- How can you decide whether your application should use multiple processes or multiple threads? Or both?

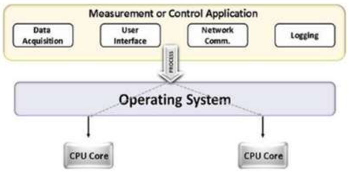
55

Multitasking vs. Multithreading



Multitasking

Multithreading

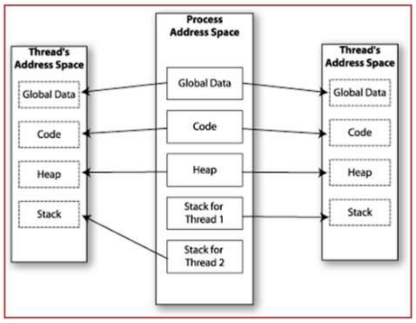


56

Multithreading vs Multitasking

- **Multithreading**
  - Threads share the same address space
    - Light-weight creation/destruction
    - Easy inter-thread communication
    - An error in one thread can bring down all threads in process
- **Multitasking**
  - Processes are insulated from each other
    - Expensive creation/destruction
    - Expensive IPC
    - An error in one process cannot bring down another process

Memory Layout: Multithreaded Program



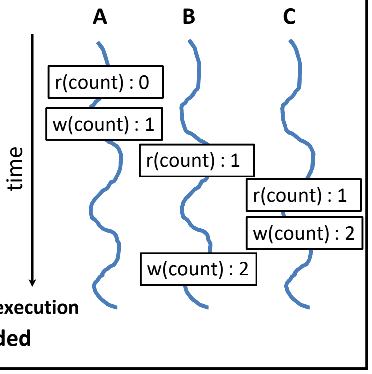
Why share the same code?
Why share the same heap?

Shared Memory

- Makes multithreaded programming
  - **Powerful**
    - can easily access data and share it among threads
  - **More efficient**
    - No need for system calls when sharing data
    - Thread creation and destruction less expensive than process creation and destruction
  - **Non-trivial**
    - Have to prevent several threads from accessing and changing the same shared data at the same time (synchronization)

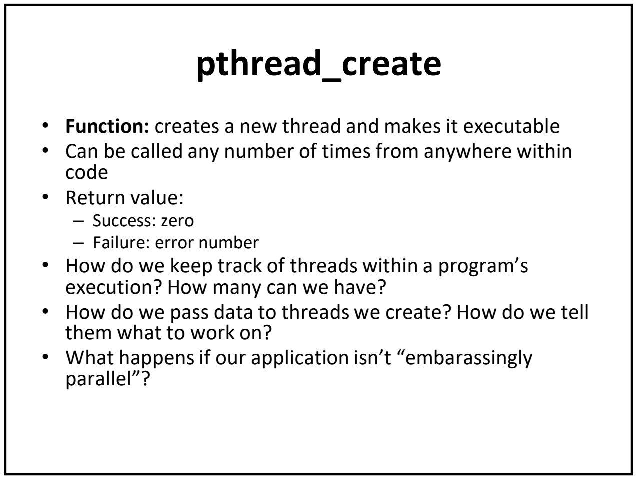
Race Condition

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



Result depends on order of execution

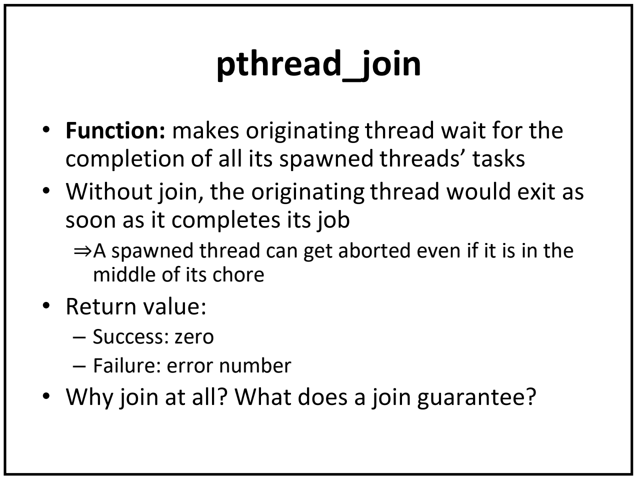
=> Synchronization needed



pthread_create

- **Function:** creates a new thread and makes it executable
- Can be called any number of times from anywhere within code
- Return value:
  - Success: zero
  - Failure: error number
- How do we keep track of threads within a program's execution? How many can we have?
- How do we pass data to threads we create? How do we tell them what to work on?
- What happens if our application isn't "embarassingly parallel"?

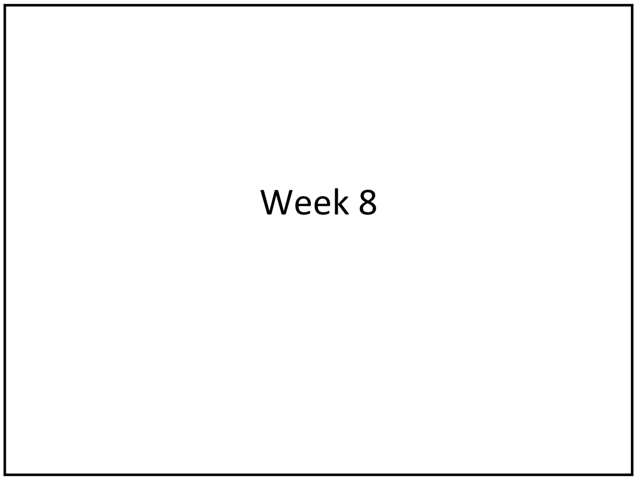
61



pthread_join

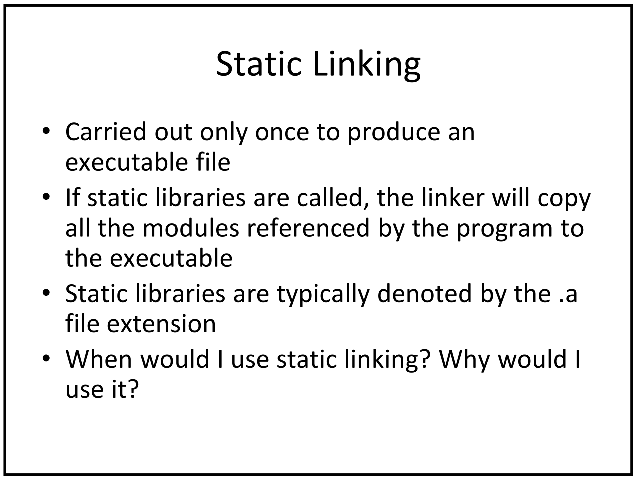
- **Function:** makes originating thread wait for the completion of all its spawned threads' tasks
- Without join, the originating thread would exit as soon as it completes its job
  - ⇒A spawned thread can get aborted even if it is in the middle of its chore
- Return value:
  - Success: zero
  - Failure: error number
- Why join at all? What does a join guarantee?

62



Week 8

63



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
- When would I use static linking? Why would I use it?

64

Dynamic Linking

- Allows a process to add, remove, replace or relocate object modules during its execution.
- If shared libraries are called:
  - Only copy a little reference information when the executable file is created
  - Complete the linking during loading time or running time
- Dynamic libraries are typically denoted by the .so file extension
  - .dll on Windows
- When would I use dynamic linking? Why would I use it?

65

How are libraries dynamically loaded?

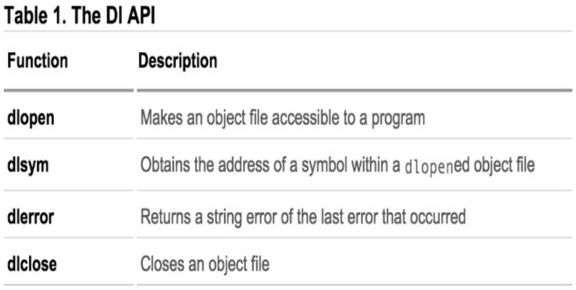


Table 1. The Dl API

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

66

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
- What other advantages are there of dynamic linking?

67

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…

68

Week 9

69

Communication Over the Internet

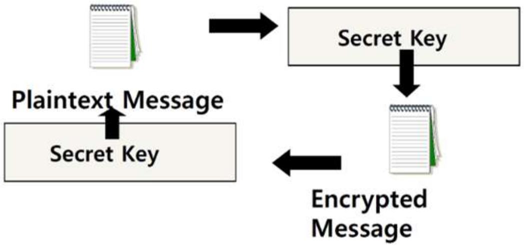
- What type of guarantees do we want?
  - **Confidentiality**
    - Message secrecy
  - **Data integrity**
    - Message consistency
  - **Authentication**
    - Identity confirmation
  - **Authorization**
    - Specifying access rights to resources

- Why do we want these guarantees?

70

Secret Key (symmetric) Cryptography

- A single key is used to both encrypt and decrypt a message



Secret Key

Plaintext Message

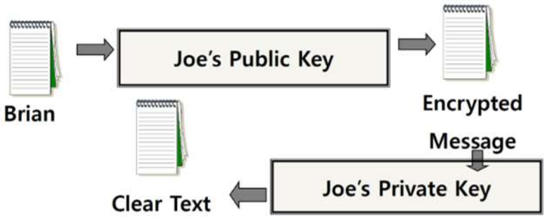
Secret Key

Encrypted Message

71

Public Key (asymmetric) Cryptography

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



Brian

Joe's Public Key

Encrypted Message

Joe's Private Key

Clear Text

72

Encryption questions

- Why have encryption?
- What are the differences between symmetric and asymmetric encryption? When would I use one or the other?
- What is used on the Internet? What is a certificate authority?
- How can I trust a message came from someone?

73

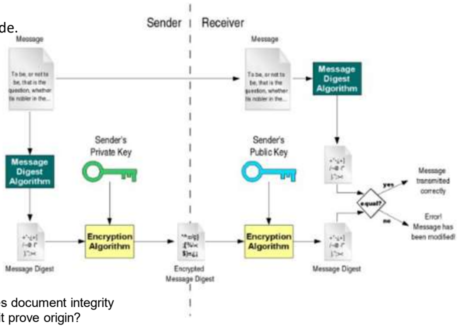
Digital Signature

- An electronic stamp or seal
  - almost exactly like a written signature, except more guarantees!
- Is appended to a document
  - Or sent separately (detached signature)
- Ensures data integrity
  - document was not changed during transmission
- How are signatures different than encryption?

74

Digital Signature

You should understand this entire slide.



- Verifies document integrity
- Does it prove origin?
- Who is Certificate Authority (CA) ?

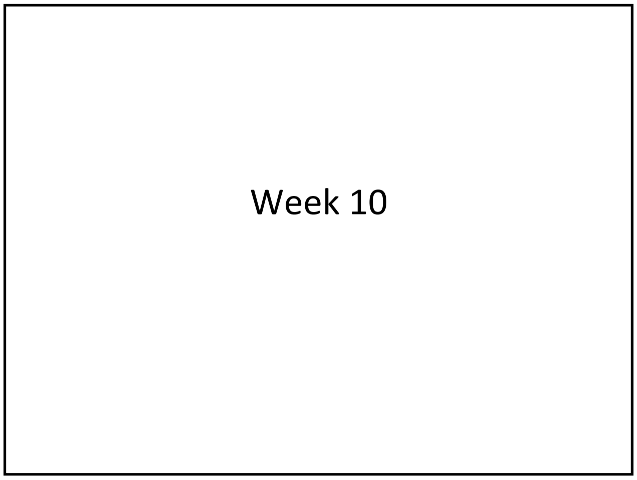
Image Source : gdp.globus.org

75

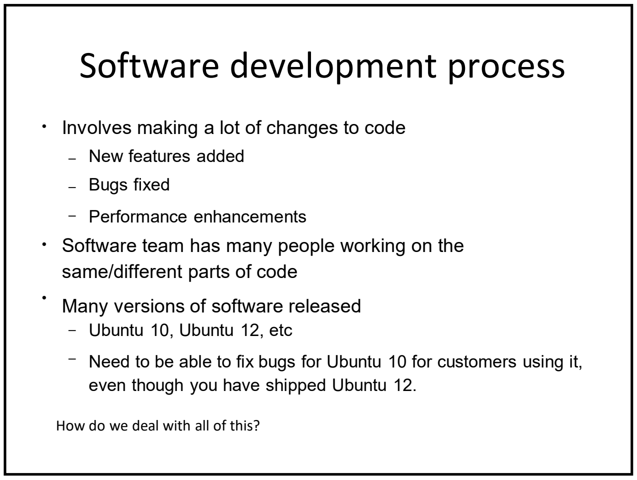
Detached Signature

- Digital signatures can either be *attached* to the message or *detached*
- A detached signature is stored and transmitted separately from the message it signs
- Commonly used to validate software distributed in compressed tar files
- You can't sign such a file internally without altering its contents, so the signature is created in a separate file
- Why detach at all? Why are signatures useful?
- Who can create a signature? How do I verify a signature?

76



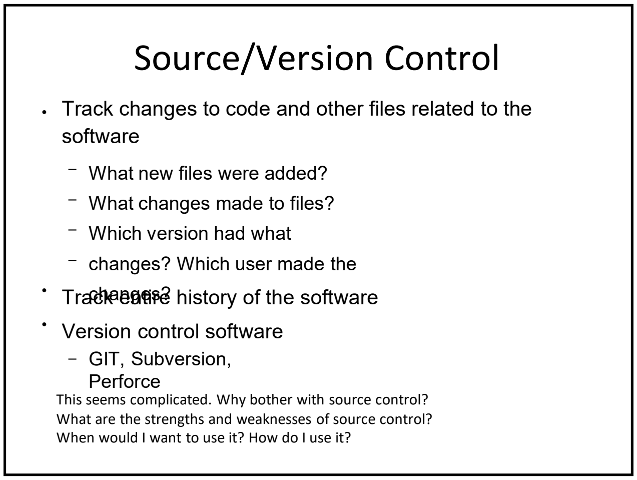
Week 10



Software development process

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

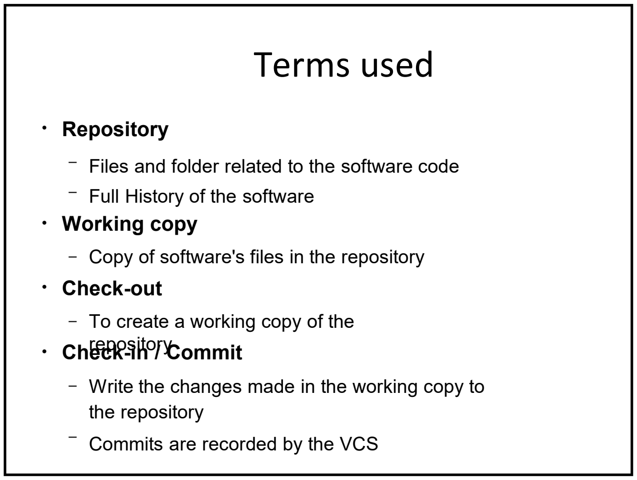
How do we deal with all of this?



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

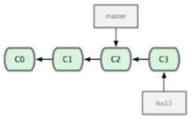


Image Source: git-scm.com

81

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

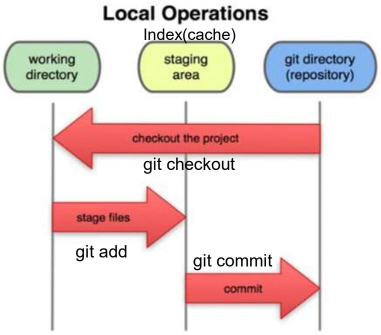
82

Questions

- What is the difference between a working copy and the repository?
- What is a commit? What should be in a commit?
- What is HEAD and HEAD^2?
- 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?

83

Git States



Index(cache)

git checkout

git add

git commit

Image Source: git-scm.com

84

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
  - $ git log files) (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.

85

Good luck for the finals week!

86