Contents

1 File Systems

Most often thought of as a tree structure
Can be a DAG structure (With hard/symbolic links)
Provides a mental model of how the directory is structured. Thus, it is easier to understand why some commands (e.g. `mv`) are fast while others (e.g. `sort`) are slow.

1.1 Why `/bin` and `/usr/bin`?

File systems at the time were so small, we couldn't fit all commands into one directory. More specifically, when booting up, people wanted to have a small set of commands that always worked and a larger set of commands after successfully booting. Now, we have both `/bin` and `/usr/bin` for backwards compatibility where `/bin` is a symbolic link to `/usr/bin`.

1.2 Implied Directories

`/` Root directory

> **Note:** `.`, `..` for `/` are both `/` itself

`.` Current directory

`..` Parent directory

1.3 Navigating the File System

File name components cannot be empty and can contain any characters **except**: `/`
Links map file name components to files.
Think of a directory as a list of file name components.
Using `namei("/usr/bin/diff")` will point to the inode of the file path.

Note: `/usr///bin/////sh` is equivalent to `/usr/bin/sh` **Note:** `/usr/bin/sh/` **must** be a directory. If it's not, it throws an error.

1.3.1 How the Operating System Finds Files (`namei()`)

```
if (f[0] == '/')
    p = root directory's ID
else
    p = current directory's ID
```

where `p` returns the file in the directory that `p` points to. When it encounters a symbolic link, it replaces the path with the contents of the symbolic link (see **Symbolic Links**).

Note: This is what the OS does. The shell will perform variable expansion when running programs.

1.4 File Permissions

Read/Write/Execute permissions have 10 bits in the format `drwxrwxrwx`

1	`d` if directory, `-` if normal file, `l` if symbolic link
2, 3, 4	`read, write, execute` permissions for user(owner) of the file
5, 6, 7	`read, write, execute` permissions for group of the file
8, 9, 10	`read, write, execute` permissions for other(world) of the file

1.5 Read/Write/Execute Bits

`-`: Flag is not set
`r`: File is readable
`w`: File is writable/can be created or removed (if directory)
`x`: File is executable/listed (if directory)
`s`: Set group ID (`sgid`); For directories, files created will be associated with the same group as the directory rather than the user. Subdirectories will inherit `sgid`

1.6 Modify Permissions (`chmod`)

`chmod [OPTION] MODE FILE`
MODE Format: `[who][op][permission]` where

who: `ugoa` (user, group, other, all)

op: `+, -` (add, remove)

permission: `rwx` (read, write, execute)

1.7 Reclaiming Storage

When there are 0 links to a file, the operating system will "reclaim" that storage, but does not actually reformat/erase the storage associated with the file. Instead, it will put that storage into "free" memory, where it can be overwritten without warning. Consequently, this means that it is possible to still see the contents of the file by reading the bytes that are on the physically on the drive.
Note: To do a better "remove", the command `shred FILE` will scramble, inject, delete bits at random to attempt to "shred" the file contents.

2 Links

A link maps a file name component (see **File Name Components**) to a file. By default, there is a minimum of 1 hard link to a file: it's file name component.

Note: It is possible to have a file with 0 hard links. For example, consider

```
(rm file, cat) <file
```

The code above will remove the hard link to `file`, but `cat` has its standard input set to `file`, so it is still open. You can still read and write to it, so the OS will not reclaim the storage from `file` until no one can access it. As soon as the program above terminates, no one can access `file`, thus the OS will reclaim the storage from `file`.

2.1 Hard Links

Hard links are like pointers to the inode of the file.

Example: Let `foo` be a file. Using the link command (see **Link**) `ln foo bar`, `bar` is a hard link to `foo`, meaning it points to the same inode that `foo` does.

Note: Hard links **cannot** point to directories. The reason for this is because then the directory structure will go from a tree/DAG to an arbitrary graph with cycles. Then, certain operations such as recursive operations will never terminate.

2.2 Symbolic/Soft Links

Symbolic links are a special file that contains the file path (**relative to the directory it is in**) of the file it wants to point to. Because the contents of symbolic links are strings, they can be altered.

Example: Let `foo` be a file. Using the link command (See **Shell Commands**) `ln -s foo bar`, `bar` is a symbolic link to `foo`, meaning it is a special file that contains the file path of `foo`, and not the actual file `foo` points to.

Note: Symbolic links **can** point to directories.

Note: While regular symbolic links are relative to their path, absolute symbolic links (ones who's contents start with `/`) are **not** relative to their path, but are absolute file paths starting at the root directory.

Note: There can be dangling symbolic links that contain a path to a file that does not exist. **Note:** You can have symbolic link loops but the shell will output errors after a while.

2.3 Link Examples

Let `foo` be a file. Then, we can have

```
ln foo bar
ln -s bar baz
ln baz buz
```

Then, when we do `ls -l foo bar baz buz`, we get

```
-rw-r--r--  3 user group  0 Feb  7 21:50 bar
lrwxr-xr-x  1 user group  3 Feb  7 21:50 baz -> bar
-rw-r--r--  3 user group  0 Feb  7 21:50 buz
-rw-r--r--  3 user group  0 Feb  7 21:50 foo
```

where `bar` is a hard link to `foo`, `baz` is a symbolic link to `bar`, and `buz` is a hard link to `baz`. Note that the hard link count for `baz` does not get incremented to 2 when we link `buz` to `baz`, but the hard link counts of `foo, bar, buz` get incremented to 3.

3 Shell

The `shell (sh)` is a lightweight scripting language that wraps the operating system

Provides an interface for the OS

Also called a Command Line Interface (CLI)

Has **no** reserved words

3.1 Exit Status (`$?`)

Prints the previous command's exit status (denoted as `$?`)

0: Successful

$\neq 0$: Error

3.2 Control Structures

`!` The not operator

Because there are no reserved words in the shell, the following code is possible

```
if=27
if[$if=27]
then
...
```

3.2.1 Brackets, Braces and Parentheses (`[ ]`, { }, `( )`)

`[ ]` Are for comparisons

{ } Run the commands in place and ignore exit statuses. It will also treat the string of commands as one command

Note: This means that if you change directories during a command inside { }, your cwd will also change

`( )` Run the commands in a subprocess and ignore exit statuses. It will also treat the string of commands as one command

Note: This means that if you change directories during a command inside ( ), your cwd will not change

Examples

```
{cd /; ls -l;} && cmd
```

will run both `cd` and `ls` regardless of their exit statuses, and the cwd will now be `/`

```
(cd /; ls -l;) && cmd
```

will run both `cd` and `ls` regardless of their exit statuses, and the cwd is still (assuming you were at before the subprocess

```
if [$a = $b]
then
  ...
else
  ...
```

is a simple `if then else`

```
if if [$a = $b]
     then
       ...
     else
       ...
then
  ...
else
  ...
```

is a nested `if then else`

3.2.2 Conditionals (`if, then, else, fi`)

Conditionals are written similarly to C conditionals.

Example: Consider the following conditional statement

```
if cmd1
then cmd2
else cmd3
fi
```

where `if` opens conditional, `then` is run if `cmd1` is true, `else` otherwise, and `fi` closes the conditional.

Logical comparisons are allowed:

```
cmd1 && cmd2
cmd1 || cmd2
```

where `&&` is logical **and**, `||` is logical **or**

3.2.3 Loops (`while, do, done`)

Loops are written similarly to C loops (but we only have `while`).

Example: Consider the following loop

```
while cmd1
do
    cmd2
done
```

where `while` opens the loop and evaluates `cmd1` at every iteration, `do` opens the body of the loop, and `done` closes the body of the loop.

3.3 Pipelines(`|`)

Pipelines (denoted as `|`) chain multiple shell commands.

Example 1: Consider the command

```
cat foo bar | tr a b | sort
```

where `stdout = stdin` between pipes.

Example 2: Consider the set of equivalent commands

```
cat foo bar >t1
tr a b <t1 >t2
sort <t2
```

where `t1, t2` are temporary files.

3.3.1 Differences

Example 2 runs sequentially, whereas pipelines run in parallel

Example 2 requires temporary files, whereas pipelines create a small buffer that will hang if full

Pipelines use **controlled parallelism**

3.3.2 Broken Pipes

Pipes can be broken in two ways: A command is writing to a pipe no one is reading from, and a command is trying to read from a pipe no one is writing to.

Example: Consider the command

```
cat foo | head -2
```

`head` will terminate after it receives 2 lines, but `cat` will still be writing to a pipe. This creates a broken pipe because `cat` is writing to a pipe no one is reading from. By default, the shell will kill `cat`

Example: Consider the command

```
echo foo | ls
```

This is not really an issue because `foo` will just put all its contents into the buffer and terminate, and `ls` will just list the entire buffer.

4 Shell Expansion

Let `var='a b c'` be a variable in a shellscript or terminal. `var` can be expanded to its contents by typing `$var`.

Note: Shell has no reserved words
Note: Anything after `--` in a command is treated as a file

```
touch ./-rf
rm -- -rf
```

will remove the file `-rf`

4.1 Predefined Variables

`$?` Exit status of the last command

`$1, $2, ...` Parameter number (where `$0` is the name of the command)

`$*` Equivalent to `$1, $2, ...`

`$var` Expands `var` to `a b c`

`$$` Expands to the process ID of the shell itself

`$!` Expands to the process ID of the last background process

`cmd&` runs `cmd` in the background

`~` or `$HOME` Expands to the home directory

4.2 Variable Expansion

Let `x='a b c'`

`${x}y` Expands to a b cy

`${x-default}` Expands to `x` if defined, `default` otherwise

`${x+set}` Expands to `set` if `x` is defined, `''` otherwise

`${x?}` Expands to `x` if `x` is defined, **error** otherwise

`${x=default}` Sets `x` to `default` if not defined, `x` otherwise

`${x:-default}` Expands to `x` if nonempty, `default` otherwise

`${x:-nonempty}` Expands to `nonempty` if `x` is nonempty, `''` otherwise

`unset x` Uninitializes `x`

`export $x` Exports the value of `x` to subcommands

4.3 Tilde Expansion

Let `x=~`

`~` Expands to the home directory

`~ name` Expands to `name`'s home directory

Examples

```
x=~
echo $x
```

will expand to `~` → `home directory`

4.4 Command Substitution

If there is a `$(abcd)`, the shell will run the command and replace the expression with the output

Example

```
grep abc $(find * -name '*.c')
```

will evaluate `$(find * -name '*.c')` before running `grep abc` on those files

4.5 Arithmetic Expansion

`$((x+5))` does arithmetic in shell

4.6 Field Splitting

Let `x='a b c *.c'` and consider the following

```
grep foo $x
```

will expand to `grep foo a␣b␣c␣*.c` where each file is a separate argument

4.7 Globbing

Similar to regular expressions (see **Basic `grep`**).

4.7.1 Globbing Syntax

`*` Matches anything (equivalent to `grep .*`)

Note: The shell will expand `*` before `grep` sees it

Note: `*` will **not** match leading `.`'s

`?` Matches any single character (equivalent to `grep .`)

`[ ]` Exactly the same as `grep [your-regex-here]` **except**, we use `!` instead of `^`

Example

Let `x='a b c *.c'` and consider the following

```
grep foo $x
```

will expand to `grep foo a␣b␣c␣*.c` → `grep foo a␣b␣c␣d.c␣e.c␣...` where each file is a separate argument

4.8 I/O Redirections (<, >, 2>, >>)

I/O Redirections redirect standard input, standard output, and standard error.

`<` set standard in

`>` set standard out

`2>` set standard error

`>>` append standard out

`&-` close

Note: IO redirects will overwrite an existing file. To prevent overwriting a file without warning, use `set -o noclobber`.

Example: Consider the command

```
cat foo <file0 >file1   2>err
        stdin  stdout   stderr
```

where `<` sets standard input to `file0`, `>` sets standard output to `file1`, and `2>` sets standard error to `err`.

Example: Consider the command

```
cat foo        >bar     2>&1
        stdin  stdout   stderr
```

This sets `bar` to standard output and standard error also to `bar`. There is only one channel into `bar`

Example: Consider the command

```
cat foo        >bar     2>bar
        stdin  stdout   stderr
```

This sets `bar` to standard output and standard error creates a second channel to `bar`. So, stdout and stderr will compete and overwrite each other.

Example: Consider the command

```
cat foo        >bar     2>&-
        stdin  stdout   stderr
```

This closes stderr

5 Basic `grep` (Global Regular Expression Print)

5.1 Overview

Grep is used to search for regular expressions. You can use `gp (global print)` to print all lines in a file, or `g/your-regex-expression/p` to search for a specific expression.

5.2 Basics

5.2.1 Basic Syntax

`^` Only special if you specify the start of the line unless it is inside brackets (see **Bracket Syntax**)

`$` Only special if you specify the end of the line

`[ ]` Match any occurrence of a single character that is between the brackets

`\( \)` Treats anything contained in the parentheses as one pattern (most often used with `*`)

`*` Match one or more occurrence to the character immediate left of `*` or the contents inside of `\( \)`

`.` Matches all characters **except** newline

\ Escape the special characters in this list

Examples

```
grep 'abc'
```

reads from standard input then finds and prints any matching occurrences of `abc`

```
grep usage diff.c
```

reads from `diff.c` then finds and prints any matching occurrences of `usage`

```
grep '' /etc/passwd
```

searches for any occurrences of `\n` (newline) in `/etc/passwd`

```
grep @ /etc/passwd
```

searches for any occurrences of `@` in `/etc/passwd`

5.3 Options

`-n`: Print line numbers

`-l`: Print file names

`-v`: Print nonmatching line numbers

`-i`: Ignore case

5.4 Brackets (`[ ]`)

Brackets are used to match any occurrence of a single character that is between the brackets.

Note: Grep reads left to right and searches greedily.

5.4.1 Bracket Syntax

`^` Must be at the front. Finds occurrences that do **not** match what follows

`-` Range operator. When put at the end, `grep` searches for `-` itself

`\[` and `\]` Escapes the brackets, so it searches for the brackets themselves

Examples

```
grep [aeiou]
```

searches for any occurrences of `a`, `e`, `i` `o`, or `u`

```
grep [aeiou]"
```

searches for any occurrences of `a"`, `e"`, `i"` `o"`, or `u"`

```
grep [a-z]
```

searches for any occurrences of any lowercase letter

```
grep [a-z0-9]
```

searches for any occurrences of any lowercase letter or number between 0 and 9 inclusive

```
grep [a-z0-9-]
```

searches for any occurrences of any lowercase letter or number between 0 and 9 inclusive or `-`

Note: To find `-` inside brackets, put it at the very end

```
grep []a-z]
```

searches for any occurrences of `]` or any lowercase letter

Note: To find `[` inside brackets, put it at the very front

```
grep []a-z]
```

searches for any occurrences of `]` or any lowercase letter

```
grep []
```

is invalid

```
grep [^a]
```

searches for any **non**-occurrences of `a`

```
grep [^a-z]
```

searches for any **non**-occurrences of a lowercase letter

```
grep [a^]
```

searches for any occurrences of `a` or `^`

Note: To find `^`, put it anywhere **not** at the front

```
grep .
```

searches for anything that is **not** \n (newline)

```
grep "['\"\\]":
```

searches for ' " or \

6 `egrep, grep -E` (Extended `grep`)

6.1 Overview

In extended grep, parentheses `()`, the or operator `|`, and question mark `?` are special characters

6.2 Extended Grep Syntax

`^` Only special if you specify the start of the line unless it is inside brackets (see **Bracket Syntax**)

`$` Only special if you specify the end of the line

`[ ]` Match any occurrence of a single character that is between the brackets

`*` Match one or more occurrence to the character immediate left of `*`

`( )` Treats anything contained in the parentheses as one pattern (most often used with `*`)

Note: `)` is only special when paired with an opening `(`

`|` Logical **or**: Let `p` and `q` be two expressions. Then, `p | q` will match either `p` or `q`

`?` Equivalent to saying `p|''`, where `p` is an expression and `''` is a newline character

. Matches all characters **except** newline

{i, j} Matches the expression anywhere from i to j times (inclusive).

Example: P{3, 5} is equivalent to PPP(PP?)? where P is a regular expression pattern.
If i is unspecified, the expression will evaluate to P{, 5} and will match up to 5 instances.
If j is unspecified, the expression will evalutate to P{3, } and will match 3 or more instances

\ Escape the special characters in this list

\# At the very end of expressions, this matches exactly what's inside the parentheses (# goes up to 9)

6.2.1 Predefined Character Sets

[:ascii:], [:alpha:], [:digit:], [:alnum:] are predefined character sets that can be used with grep to match any character in that set.

Example: Consider the following expression

```
grep [[:ascii:]]
```

will match any ascii character using the bracket operator (see **Brackets**)

Examples

```
grep -E "(^|[^/])\*([^/]|$)"
```

will match any occurrence of * that isn't a comment

```
grep -E "a(b*c)d\1"
```

equivalent to grep -E "a(b*c)d(b*c)"

```
grep -E "(a*)(b*)c\2\1"
```

equivalent to $a^n b^n c b^n a^n \ \forall \ n \in \mathbb{Z}^+$

7 Scripting Languages (Elisp, Python, JavaScript)

There are different types of scripting languages

Wrappers (e.g. shell)

Embedded/Extension Languages (e.g. ELisp)

Object Oriented/Packaged Languages (e.g. Python)

7.1 Elisp

Developed in 1950's

Core CS language (functional language)

Used in AI

7.1.1 Data Types

Numbers: int, float **(no overflow)**

Symbols/Atoms: Piece of data that has a name **(singletons)**

String: Has no line boundaries

Pairs: Piece of memory with two parts **(basis for linked lists)**

nil ≡ ()

Tagged Pointers: Pointers have tags for big/small datatypes for efficiency

7.1.2 Calling Functions

Anything inside parentheses `(function arg1 arg2 ...)` is a function call

Note: The syntax is the same for functions and data

7.1.3 Writing Data

Use an apostrophe `'` prefixed to an expression to represent it as data

```
(a e g)
```

represents a linked list where `[a | ]` → `[e | ]` → `[g |Ø]`

```
(let ((a '(f e g))))
```

7.1.4 Linked Lists

```
(cons a b)
```

constructs a new pair `[a | b]`

```
(car P)
```

returns the head of the pair `P`

```
(cdr P)
```

returns the tail of the pair `P`

Note: Both `car` and `cdr` are fast operations, whereas `cons` is slower because it is similar to `new` in `C++` or `malloc()` in `C`

Examples

```
(let ((foo ((cons 29 '(39 -6))))
    (cons (cdr foo) (car foo))))
```

`foo` is the linked list `[29 | ]` → `[39 | ]` → `[-6 |Ø]`.
From that, we create a list from `foo` to get `[39 | ]` → `[-6 | ]` → `[29 |Ø]`

7.1.5 Improper Lists

Lists that are not null terminated

```
(a . b)
```

creates an improper list of contents `[a | b]`

Examples

```
(19 27 32 . 14)
```

is the improper list `[19 | ]` → `[27 | ]` → `[32 |14]`

7.1.6 Tree Structure of Nested `car/cdr`

Consider the following improper list

```
v = ((37 -6) . 29)
(car (cdr (car v)))
```

is equivalent to

```
(car (cdr (car v)))
= (car (cdr (37 -6)))
= (car (-6 nil))
(car (cdr (car v))) = -6
```

will return `-6`

7.1.7 Functions

Elisp uses prefix notation

```
(let ((v1 E1)
      (v2 E2)
        ...
      (vn En)))
```

binds v_i to E_i $\forall\ 1 \leq i \leq n$

```
(function a b ...)
```

is equivalent to `function(a, b, ...)` in C++

```
(+ a b)
```

is equivalent to `a + b` in C++

7.1.8 Control Statements

```
(if A B C)
```

"if `A` then `B` else `C`"

```
cond ((E1 E2)
      (E3 E4)
        ...
      (E(n-1) En))
```

"if `E1` then `E2`, else if `E3` then `E4`, else if ... else if `E(n-1)` then `En`

7.1.9 Defining Functions and misc.

See Lisp Reference

7.2 Python

An interpreted language (runtime checking) that reads like pseudocode

Note: Python relies on indentation. `TAB` and `4*SPACE` are different

7.2.1 Objects

Every Python object has the following properties:

An identity/address (immutable)

A type (immutable)

A value (mutable **iff** object itself is mutable)

Associated with objects are attributes (private variables) and methods

`obj.name` is `name` of `obj`

`obj.method(args...)` is a function call to `method()` on `obj`'s behalf

7.2.2 Built-in Functions

`id(obj)` returns the identity of `obj` as an integer

`type(obj)` returns the type of `obj`

`a is b` compares identities

`a == b` is a logical equality comparator

`isinstance(obj, class)` true if `obj` in `class`, false otherwise

7.2.3 Types

`NoneType`: Equivalent to `nullptr`

Numbers: int, float, complex (`a + bj`), boolean (`0, 1`)

Sequences: see **Sequence Type**

Mappings: see **Map Type**

Callables: Functions, methods, classes

Internal Types: etc.

7.2.4 Sequence Type

Operations on sequences include the following

Indexing: `seq[i]` will compute the $\texttt{i}^{\text{th}}$ element of the sequence (if $\texttt{i} < 0$, it becomes `len(seq) +` $\texttt{i}^{\text{th}}$ element)

Length/Size: `len(seq)`

Subsequence: `seq[i:j]` where `i, j` < `len(seq)`

Note: Range is `[i, j)`
If `i` is not specified, the subsequence starts at the beginning of the sequence
If `j` is not specified, the subsequence goes from `i` to the end of the sequence

Lists (`ls = []`) are mutable sequences. Common functions include

`append()` C++ vector push_back() or emplace()

`extend()` joins two lists together

`insert(i, e)` inserts an element `e` at index `i`

`pop(i)` pops the element at index `i` (default is to pop from the back of the list)

`reverse()` reverses the list

`sort()` sorts the list

7.2.5 Map Type

Dictionaries (`map = {'a' : 1, 'b' : xyz}`) are like hashmaps or unordered_maps. Common functions include

`has_key(k)` returns true if the key `k` exists, false otherwise

`get(k)` returns the value at key `k` if it exists, `None` otherwise.

`del map[k]` deletes the dictionary entry at key `k`

7.2.6 Callables

```
def func(x, y):
    return x + y + 1
```

defines `func`

```
func = lambda x, y: x + y + 1
```

defines a lambda function to `func`

```
func(y = 2, x = 1)
```

will set `x = 1, y = 2` even if the parameters themselves out of order

Note: Python allows for explict definition of parameters

7.2.7 Classes

Like C++ classes but `this` is now `self` and is explicit within the class

```
class c(a, b):
    __init__(self, ...):
        constructor definition here
    def hello(self, a, b):
        return self.x + a + b
```

Work like functions in the sense that you can say `d = c` and do `obj = d()`

7.2.8 Namespaces

By definition, a class is an object, thus it has a namespace. Namespaces act similarly to C++ namespaces.

`__dict__()` returns all of your namespace components in dictionary form.

Example

```
c.__dict__() = {'hello': method, 'x': object, ...}
```

calling `__dict__()` on the class `c`

7.2.9 Modules and Packages

Note: `import` is declarative
When we import modules into our python file, the following happens

A new namespace `N` is created

Reads and executes all the code under the module in N

Bind the module name to `N` in the invokers context

This means we can now do `module.func()` assuming we did `import module`

To selectively import parts from modules, we do `from module import your-parts-here`

Note: `from module import *` is equivalent to `import module`

Packages are just modules in directories. To import, we do `import path.to.your.mod.module` where `.` is equivalent to `/` in the shell

Note: Packages usually have an empty `__init__.py` file

7.2.10 Why Packages and Classes?

Packages are more for the software developers, while classes are for behavior of objects at runtime.

7.2.11 Entry Point

`__name__ == "__main__"` is the entry point for the top level code (imported modules don't execute `"__main__"`).

7.2.12 Virtual Environments

A virtual environment lets you run different versions/configurations of Python. This is useful for portability and compatibility. To create a venv, we run

```
python3 -m vemv mydir
. bin/activate
```

Now, the virtual environment is technically set up. All that's left is to `pip install` whatever you need to create your venv

7.3 JavaScript

Developed in 10 days

Similar to Python

Scripting language → forgiving

Can be hooked into HTML

7.3.1 Hooking to HTML

```
<script src = 'script.js'></script>
```

will load `your-webpage-here/index.hml` then `your-webpage-here/script.js`. Note that this takes 2 get requests. For smaller programs, we can write `<script> your-script-here </script>` directly in the HTML.

7.3.2 Protecting Your Code

Program Obfuscation: Turning good, readable source code to shit. It'll proably reduce the size of your file and may deter regular people, but genearlly, it doesn't work (we have deobfuscators).

Don't Ship Scripts: Simply don't ship your scripts to the client.

7.3.3 JSX (JavaScript eXtension)

```
const n = <p style = 'your-style-here'> ... </p>;
```

where `const n =, ;` are JS while `<p style = 'your-style-here'> ... </p>` is HTML

7.3.4 Order of Execution

Browser Rendering Pipeline

(1) Browser downloads the HTML webpage and may try to render before it's finished downloading. The problem with this is that your browser may need to rerender objects that depend on unreceived packets. Moreover, if your browser tries to run scripts that rely on unreceived data, they'll crash.

(2) Optimization: Your browser will prioritize the elements that are on the screen

7.3.5 JSON (JavaScript Object Notation

Another competitor to XML

```
{
  "menu": {"id": "file",
           "value": "foo",
           "popup": {"menuItem: [array, of, elements]}
          },
 "plate": "your-value-here",
 "napkin": "your-other-value-here"
}
```

creates a set of key-value pairs for your HTML webpage

7.3.6 NodeJS

JS runtime for asynchronous events **Event Handler Paradigm:** Write your program as a set of event handlers

```
while(g = getEvent())
  handleEvent(g)
```

is predefined by Node. We just need to write the `handleEvent(g)` portion

Event Handlers

Must be fast

Must return

Cannot wait (split long operations up)

NodeJS is single-threaded, meaning at most one event handler can be active at any given time. How do we scale then? Multiprocessing: Run multiple web servers/pages

7.3.7 Multithreaded Applications

With multithreaded applications, we can use parallelism where multiple threads run different operations in parallel. However, it get's buggy really quickly due to race conditions: 2 threads read/write to shared memory which can end up in a deadlock quickly.

7.3.8 Multithreading vs Multiprocessing

Multithreading: Multiple threads of a process are executed at once

Multithreading: Every thread uses shared memory

Multiprocessing: Multiple processes are executed at once

Multiprocessing: Every process uses distinct memory

8 Client-Server

8.1 Overview

The client-server structure states that an application is partitioned into a server and client side. Clients send request to the server(s) and get a response. But, if the server goes down, all the clients can't use the application.

8.2 Alternatives to Client-Server

Peer-to-Peer: The application is partitioned across the network where no node is more important than another.

Pros: One goes down? Others stay up

Cons: More overhead: Everyone needs to constantly talk to each other

Primary-Secondary: We have a primary controller and secondary worker bees. Primary keeps track of the state, secondary works on smaller requests

Pros: One node in charge of state

Cons: If that one node goes down, you're fucked

8.3 Performance

Two main issues:

Throughput: How much data you can send through (bottleneck threshold)

Out of Order Execution: Requests get handled out of order/in parallel to maximize efficiency

Downside: order of execution may matter

Latency: How long it takes to communicate between the client and server (delay)

Cache: Keep a cache of recent requests

Downside: Stale caches and unsynchronized applications

9 The Internet

9.1 Circuit Switching

Physical network of wires connecting devices. Therefore, if there is a path from one node to another, two people can reserve those wires when they get on the line. Once someone reserves those wires, no one can use them. This is called **Guaranteed Effort**

9.2 Packet Switching

Proposed by Paul Brown in 1961.
Strategy: Divide a piece of a message into multiple smaller packets, shipping each one individually. This is known as **Best Effort** because it isn't guaranteed that the packages will arrive in order or even arrive at all. Problems include

Packets can get lost

Packets can get receive out of order

Packets can be duplicated

Security issues (intercepted/corrupted packages)

How do we solve these issues? See **Layers of the Internet**

9.3 Layers of the Internet

Assuming you can establish a connection between two nodes A and B, we can add layers of abstraction. From bottom-to-top we have:

(1) Physical Layer: Physical wires connecting adjacent nodes

(2) Link Layer: Send packets to adjacent nodes

(3) Internet Layer: Send packets to non-adjacent nodes

(4) Transport Layer: Defines connections between the client and server, sending streams of data between A and B (stream = multiple packets)

(5) Application Layer: Clients and servers talk to each other through API's

9.3.1 Internet Protocol

We use both IPv4 and IPv6 for compatibility (we can't upgrade everyone at once). A packet can be split up into its header and its contents.
The Header pt

Length

Protocol Number (unique 32-bit identifier)

Source and Destination addresses (IP addresses)

Checksum: 16 bits to check for packet corruption

IP Addresses: are in the format `XXX.XX.XXX.X` where each set of `X`'s between the `.`'s represent a 8 bit number (0-255)

TTL (Time To Live): Every time a package gets sent, it's TTL gets decremented. When TTL = 0, the package gets thrown away (for performance)

IPv6 pt

Wider addresses (hex)

Less efficient than IPv4

Communicating between IPv4 and IPv6: Can't directly communicate with each other. Instead, we use NAT (Network Address Translation) and gateway hosts as a workaround

9.3.2 UDP and TCP

UDP (User Datagram Protocol): Thin layer above IP. Speeds up communications by not formally establishing a connection before data is transferred (mainly used for debugging and low level stuff).

TCP (Transmission Control Protocol): Built on top of IP and takes streams of data that provide transmission that's reliable, ordered, and error-checked.

Flow Control (TCP): We don't want to overload the router, so we control how many packets we send at once. Once we receive them, we want to reassemble the original message. If packets are missing or corrupted, TCP will try to retransmission said packets.

Note: Though TCP is reliable, it's slow

9.3.3 RTP and HTTP

RTP (Real-Time Transport Protocol): Builds atop UDP and is mainly used for faster but less reliable communication. Rather than wait for packets to come, it drops them and moves on (TCP wouldve been jittery because it's waiting for packets).

HTTP (Hyper Text Transfer Protocol): Builds atop TCP and is the basis for the web. Protocol is as folllows: client opens the connection on webserver and sends a request, the server responds, and closes the connection.

Side Note: `telnet` in Emacs sets a TCP connection from Emacs to leapsecond.com on port 80 (default). GET / HTTP/1.0/ tells us I want teh root document and is using HTTP protocol 1.0 (ancient).

9.3.4 HTTP (cont.)

HTTP/1.1 (1997) broke the webpage up into pieces and allowed for connections to stay open. **Side Note: HTTPS** is just HTTP + encryption (doesn't encrypt metadata)

HTTP/2.0 (2015 allowed for header compression, which puts more burden on the CPU but improves performance. Introduced server push (servers don't need to wait for clients to send requests), pipelining (multiple requests/responses can be sent at once, potentially out of order), and multiplexing (uses a single connection to satisfy several different web searches)

HTTP/3.0 (2022) uses UDP (technically QUIC which is a substitute for TCP with support for multiple streams), improved latency, and tried to resolve head-of-line delays.

Head-of-Line Delays: Say a recipient has packets 1 2 and 3. It hasn't gotten packet 4 but received packet 5, 6, 7. TCP will show packets 1 2 3 but stop at packet 4 to ask for retransmission. When packet 4 comes it will show 4,5,6,7 (head of line delay). In this case, the other packets waited for head of line (packet 4) to show up.

10 HTML

The Web = HTTPS + HTML
HTML is the stream that we're sending via packets.
Derived from SGML (Standard Generalized Markup Language)

```
<p>
  your SGML code here
</p>
```

where <p> is the node

```
<p style="your-style-here">
  your SGML code here
</p>
```

you can specify styles and other attributes

DTD (Document Type Definition): They sucked. Basically SGML templates with predefined node attributes/properties

Note: HTML 1-4 sucked bc they tried to standardize it, but the web expanded too quickly. HTML5 is most commonly used now because it adopted a **Living Standard** model, where new features get added regularly

10.1 Why HTML?

SGML sucked

Compatibility issues with web-specific extensions

It's forgiving: if you fuck up, it'll still print something. The downside? If you fuck up, it'll still print something (harder to find bugs).

10.2 DOM (Document Object Model)

Specifies the type of tree you have in your browser and how you can access/manipulate elements via API's.

Essentially tree manipulation

10.3 XML

SGML turned into XML (eXtensible Markup Language)
XML is very strict on syntax, but has no real weight. People tried XHTML which sucked

Note: XML is still used in many business-to-business applications mainly used to ship databases

Note: Implicit ending paragraph tags are allowed in HTML and SGML but not XML

11 CSS (Cascading Style Sheets)

Basic Idea: Separate form and function

CSS specifies priorities between attributes in the DOM, browser, and the user specifications

11.1 Cascading

Style at a tree node is inherited by its descendents. That is, if the root has `style = "Times New Roman"`, its descendents inherit `style = "Times New Roman"`

11.2 Style

Styles are declarative. This means that CSS is easy but restrictive.

12 Software Construction

12.1 Purpose of Applications

Survive power outages/OS bugs: How? Persistant storage

Be fast: How? Cache

Be understandable

12.1.1 Test First Paradigm

Write test/use cases before building out the framework. This will significantly reduce time spent on refactoring code.

13 Emacs

Note: We sometimes prefer non-GUI versions of programs like Emacs because of the latency: if we `ssh` into a server that is across the world, the GUI will be slower. For example, it needs to send a request, wait, and receive a response before moving your cursor. Thus, we may prefer editing straight from the terminal.

13.1 Commands

Emacs commands are structured like a tree. For example, `C-` is the entry point, `n, p, etc.` are leaf nodes, `x` is an entry to a subtree, etc.

14 Shell Commands

14.1 Change Working Directory

`cd [-L | -P] [DIRECTORY]`: Change working directory

`-L`: Handle dot-dot logically; symbolic link components are **not** resolved before dot-dot components are processed

`-P`: Handle dot-dot physically; symbolic link components are resolved before dot-dot components are processed

Note: If both `-L` and `-P` are specified, the last of the options will be used. If neither is specified, the oeprand will handle dot-dot logically

14.2 Disk Usage

`du [OPTIONS] FILE(S)`: Estimate file space usage (If no `FILE` is specified, list usage for all directories (recursively) and files)

`-0, --null`: End each output line with `NUL`, not newline

`-a, --all`: Write counts for all files, not just directories

`--apparent-size`: Print apparent sizes rather than device usage; Though the apparent size is usually smaller, it may be larger due to holes in ('sparse') files, internal fragmentation, indirect blocks, etc.

`-B, --block-size=SIZE`: scale sizes by `SIZE` before printing them; `e.g., -BM` prints sizes in units of 1,048,576 bytes

`-b, --bytes`: Equivalent to `--apparent-size --block-size=1`

`-c, --total`: Produce a grand total

`-D, --dereference-args`: Dereference only symlinks that are listed on the command line

`-d, --max-depth=N`: Recurse a max of `N` levels

`--files0-from=F`: Summarize device usage of the `NUL`-terminated file names specified in file F; if F is `-`, then read names from standard input

`-H`: See `-D, --dereference-args`

`-h, --human-readable`: Print sizes in human readable format (e.g., `1K, 234M, 2G`)

`--inodes`: List inode usage information instead of block usage

`-k`: Equivalent to `--block-size=1K`

`-L, --dereference`: Dereference all symbolic links

`-l, --count-links`: Count sizes many times if hard linked

`-m`: Equivalent to `--block-size=1M`

`-P, --no-dereference`: Don't follow any symbolic links (this is the default)

`-S, --seperate-dirs`: For directories, do not include size of subdirectories

`--si`: Like `-h`, but uses powers of 1000, not 1024

`-t, --threshold=SIZE`: Exclude entries smaller than `SIZE` if positive, or entries greater than `SIZE` if negative

`--time`: Show time of the last modification of any file in the new directory, or any of its subdirectories

`--time=WORD`: Show time as `WORD` insetad of modification time: `atime, access, use, ctime or status`

`--time-style=STYLE`: Show times using `STYLE`, which can be: `full-iso, long-iso, iso, or +FORMAT`; `FORMAT` is interpreted like in 'date'

`-X, --exclude-from=FILE`: Exclude files that match the pattern `FILE`

`--exclude=PATTERN`: Exclude files that match `PATTERN`

`-x, --one-file-system`: Skip directories on different file systems

14.3 Kill (`kill`)

`kill [OPTION] PROCESS`

`-9` Kill request cannot ignore

Note: Kill sends a request to tell a process to kill itself

14.4 Link

`ln [OPTIONS] TARGET LINK_NAME`
`ln [OPTIONS] TARGET`
`ln [OPTIONS] TARGET DIRECTORY`
`ln [OPTIONS] -t DIRECTORY TARGET`: Create a link to `TARGET` with name `LINK_NAME`

`--backup`: Make a backup of each destination file

`-b`: Like `--backup` without an argument

`-d, -F, --directory`: Allow superuser to attempt to hard link directories (will probably fail due to system restrictions)

`-f, --force`: Remove existing destination files

`-i, --interactive`: Prompt before removing destinations

`-L, --logical`: Dereference `TARGET`'s that are symbolic links

`-n, --no-dereference`: Treat `LINK_NAME` as a normal file if it's a symbolic link to a directory

`-P, --physical`: Make hard links directly to symbolic links

`-r, --relative`: Used with `-s, --symbolic`; Create links relative to link location

`-s, --symbolic`: Make symbolic links instead of hard links

`-t, --target-directory`: Specify a `DIRECTORY` in which to create the links

`-T, --no-target-directory`: Treat `LINK_NAME` as a normal file

`-v, --verbose`: Print name of each linked file

14.5 List Directory Contents (`ls`)

`ls [OPTIONS] DIRECTORY`

`-a, --all`: Do not ignore entries starting with `.`

`-A, --almost-all`: Only ignore the implied `.`, `..`

`-c`: With `-lt`: sort by, and show, ctime (time of last modification of file status information); with `-l`: show ctime and sort by name; otherwise: sort by ctime, newest first

`-f`: List all entries in directory order

`-g`: Like `-l`, but do not list owner

`-G, --no-group`: Used with `-l`, but do not list group

`-i, --inode`: Print the index number of each file (value of the pointer)

`-l`: Use long listing format

`-o`: Same as `-lG`

`-R`: Recursively list subdirectories and files

`-t`: Sort by time, newest first

`ls -l` Columns

1 File type and permissions (see **File Permissions**)

2 Hard link count

3 Owner of the file

4 Group of the file

5 Size of the file (in bytes)

6, 7, 8 Last modified date MMM DD TT:TT

9 `name` **or** `name -> contents` if symbolic link

14.6 Move

`mv [OPTIONS] -T SOURCE DESTINATION`
`mv [OPTIONS] SOURCE DIRECTORY`
`mv [OPTIONS] -t DIRECTORY SOURCE`: Move file from source to directory

`-f, --force`: Do not prompt before overwriting

`-i, --interactive`: Prompt before every overwrite

`-n, --no-clobber`: Do not overwrite existing files

`-t, --target-directory`: Move all `SOURCE` to `DIRECTORY`

`-T, --no-target-directory`: Treat `DIRECTORY` as a file

`-u, --update`: Move only when `SOURCE` is newer than or is missing `DESTINATION`

`-v, --verbose`: Explain what is being done

14.7 Remove

`rm [OPTIONS] [FILE/DIRECTORY]`: Remove a file or directory

`-f, --force`: Forcefully remove

`-i, --interactive`: Prompt before every removal

`r, -R, --recursive`: Remove directories and all their contents recursively

`-d, --dir`: Remove empty directories

`-v, --verbose`: Explain what is being done

Note: `rm` does not remove files themselves. They simply remove the hard/symbolic link to a file

14.8 Sequence (`seq`)

`seq [OPTIONS] LAST`
`seq [OPTIONS] FIRST LAST`
`seq [OPTION] FIRST INCREMENT LAST`: Print numbers from `FIRST` to `LAST`, in steps of `INCREMENT`

`-f, --foramt=FORMAT`: Use `printf` style floating-point `FORMAT`

`-s, --separator=STRING`: Use `STRING` to separate numbers (default: `\n`)

`-w, --equal width`: Equalize width by padding with leading zeroes

14.9 Stream Editor (sed)

`sed [OPTIONS] FILE(S)`: A stream editor to perform basic text transformations on inputs
Note: `sed` makes only **one** pass over input(s)

`-n, --quiet, --silent`: Suppress automatic pattern space printing

`--debug`: Annotate execution

`-e SCRIPT, --expression=SCRIPT`: Add the script to the commands to be executed

`-f SCRIPT_FILE, --file=SCRIPT_FILE`: Add the contents of `SCRIPT_FILE` to the commands to be executed

`--follow-symlinks`: Follow symlinks when processing in place

`-i[SUFFIX], --in-place[=SUFFIX]`: Edit files in place (Makes backup if `SUFFIX` is applied

`-l N, --line-length=N`: Specify desired line-wrap length

`--posix`: Disable all GNU extensions

`-E, -r, --regexp-extended`: Use extended regular expressions in teh script (for portability use POSIX -E)

`-s, --separate`: Consider files as separate rather than a single, continuous long stream

`--sandbox`: Operate in sandbox mode (disable e/r/w commands)

`-u, --unbuffered`: Load minimal amounts of data from teh input files and flush the output buffers more often

`-z, --null-data`: Separate lines by `NUL` characters

15 Version Control (git)

15.1 Overview

`git` is a version control system for software development, and is arguably the most important part of software construction. There are two main things that `git` maintains:

An object database: A repository of objects that records the history of your project

An index (cache): Records the future[1] of the project.

15.2 Getting Started

There are two main ways to start a git repository: `git init TARGET_DIRECTORY_HERE` and `git clone TARGET_REPOSITORY_HERE`

`git init TARGET_DIRECTORY_HERE` initializes an empty project inside the target directory (current directory if not specified) with a `.git` folder. This is less common, as a lot of people don't start a project from scratch.

`git clone TARGET_REPOSITORY_HERE` clones an existing repository, creating a directory on your computer containing a copty of all of the files in that repository with the `.git` folder inside that directory.

Note: When cloning a repository, it is possible to clone from a device.

Note: `git` will remember where you're cloning from; that is, if you run

[1] Future: plans for the future of the project, immediate or long-term

```
git clone REMOTE_REPOSITORY
git clone ./REMOTE_REPOSITORY
```

`git` will identify that the second clone was from a device, whereas the first clone was from a remote location.

When working with `git`, it is important to remember that remote-to-local repositories are a downstream structure; that is, cloning from a remote repository sends a repository "downstream" to your device.

15.3 The Repository

What do you put inside your repository?

Stuff you change by hand

What should you **NOT** put inside your repository?

Automatically generated files (e.g. `node_modules`)

Stuff that isn't portable/shouldn't be portable (e.g. `.env.local`)

`.gitignore`: By default, `git` creates this file, which will tell `git` to automatically ignore file/type(s) that are specified inside `.gitignore`. This file is a very important one, especially to keep your repository clean and portable.

ASIDE: Shorthand for Commit ID's

`COMMIT_ID^`: The commit before `COMMIT_ID`

`COMMIT_ID~n`: The `HEAD` - n^{th} commit

`COMMIT_ID^!`: Same as `COMMIT_ID^..COMMIT_ID`

`COMMIT_ID..COMMIT_ID`: Range of commits (`start`, `end`]

15.4 Managing the Repository

The following subsections will cover common git commands that are used to manage the repository.

Note: All of these commands are called under the assumption that you're in the current repository folder.

15.4.1 State

This set of commands gives information of the state of the repository.

git status

`git status`: Tells you the current status of your repository. Mainly, it will list all files that have been added, modified, or deleted relative to your last commit.

git ls-files

`git ls-files`: Lists all working files managed by `git` to `stdout`. Files that are not tracked by git will not show up on this list(hence why we do not just use `ls`).

git blame

`git blame`: Returns a line-by-line history of a specified file in a specified commit (`HEAD` if not specified) with the author and timestamp of each line.

git diff

`git diff COMMIT_A..COMMIT_B`: Takes a `diff` of two commits and prints to `stdout` **(See `git log` for navigating the Terminal output)**.

git grep PATTERN

`git grep PATTERN`: Same as doing `grep PATTERN $(git ls-files)` **(See `grep`)**

git log

`git log [OPTIONS] (start-point..end-point]`: Prints the commit history between `start-point` exclusive to `end-point` inclusive in reverse-time order (new → old). Prints the entire commit history from the first commit to the most recent commit if no `start-point, end-point` are specified.

Options

`-n`: Look at the HEAD - n^{th} commit

`--decorate[...]`: Format git log output with specified parameters **(See HW4)**

Navigating `git log` in the Terminal

`/PATTERN`: Searches for a pattern in the output.

`n` and `N`: Goes to the next and previous n^{th} occurrence respectively

`q`: Exits the log output

`SHIFT-g`: Scrolls to the very bottom of the log output

ASIDE: Using `git log`

`git log` is commonly piped into other commands such as `wc` **(See `Shell Commands`)**

`git log` outputs both the committer and author of a commit. While often times they are the same person, it may be that they are not. This is more apparent in big open source projects with controlled/reviewed commits. The person with repository access will be listed as the committer, while the person who wrote the code will be listed as the author.

15.4.2 Pushing Upstream

This set of commands relates to pushing upstream to the central repository, mainly staging, committing, and pushing.

git clean

`git clean`: Removes all untracked files from the repository.

git add

`git add FILE`: Stages a file to commit. If the file was previously untracked, `git` will now track the file.

git rm

`git rm FILE`: Removes file as well as untracks the file that was removed. This is equivalent to doing `rm FILE` **(See Shell Commands)** followed by `git add FILE`.

Options

`-f, --force`: Forcefully remove file and ignore any warnings

`-r`: Recursively delete a directory and all of its contents

`git reset`

`git reset [OPTIONS]`: Unstages all modified files.

Options

`--soft`: Only reset `HEAD`

`--hard`: Reset `HEAD`, the index, and working tree

`git commit`

`git commit [OPTIONS]`: Creates a commit with all of your staged files and allows for a commit message.

Commit Semantics A commit message should explain **why** they are adding to the repository, not what they are contributing. A commit message should have the following format:

```
brief summary here

* more
* details
* here
```

Options

`-m`: Write a commit message inline

`--amend`: Amend a previous commit.

Note: Amending should be done sparingly and never in big open source projects to avoid confusion.

`git push`

`git push [OPTIONS]`: Pushes all commit(s) from your local repository upstream into the central repository.

Options

`-u, --set-upstream`: Set upstream branch to push to

`--atomic`: Request atomic transaction on remote side

15.4.3 Pulling Downstream

This set of commands relate to pulling from the upstream repository, mainly fetching and pulling.

`git fetch`

`git fetch [OPTIONS] [BRANCH]`: Fetches metadata from a remote branch, `origin` if not specified. Note that all of the working files in the local repository remain unchanged. This is generally a safer way to update your local repository with the lastest metadata since it does not change any working files.

Options

`--all`: Fetch information from all remotes

`--atomic`: Request atomic transaction on remote side

git pull

`git pull [OPTIONS] [BRANCH]`: Pulls metadata from a remote branch, `origin` if not specified. Note that all of the working files in the local repository will be updated to match the upstream repository. If the local branch is behind the remote, the local branch will fast forward by default. If there are divergent branches, use either the `--rebase` or `--no-rebase` option to resolve conflicts. `git pull` will fail if there is no specified method of resolving conflicts since `git` is conservative[2]. `git pull` is equivalent to `git fetch` followed by `git merge` or `git rebase` depending on default configurations.

Options

`--all`: Fetch information from all remotes

`--atomic`: Request atomic transaction on remote side

15.4.4 Branch Manipulation

15.4.5 Overview

A `branch` is a lightweight **moveable** pointer[2] to a commit. By default, when creating a repository, there is only one branch, `main/master`. By default, when creating a new branch, `git` will branch off of the current branch. `git` is a tree structure, meaning it must be a `DAG` in order to work.

git branch

`git branch [OPTIONS] [BRANCH]`: Lists all of the repository's local branches.

Options

`-d`: Delete the branch `BRANCH`

`-D`: Delete the branch `BRANCH` without warning

`-m`: Renames a branch from `A` to `B`

git checkout

`git checkout [OPTIONS] BRANCH`: Changes all of the working files to be identical to the ones in the specified `BRANCH`. Alternatively, `git switch BRANCH` is similar but has a few minor differences.
When checking out, `git` is conservative[3] and will prevent a checkout if you have uncommitted or untracked working files.

Options

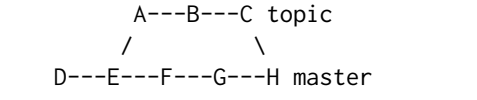
`-f, --force`: Force a checkout and ignore any warnings

`-b`: Create a new branch `BRANCH` and start it at the `start-point` of the `main` branch

`-B`: Resets `BRANCH` to a specified `start-point` if the branch exists, same as `-b` otherwise

git merge

`git merge [BRANCH]`: Merges branch `BRANCH` into the current branch. This creates a graphical commit history.



[2]Pointer: In `git`, `HEAD` is a reference variable that points to the tip of the current working branch.

[3]Conservative: `git` will warn you if you have uncommitted or untracked files when performing any actions that mutate your working files.

git rebase

`git rebase [BRANCH]`: Reapplies commits atop a branch tip. This creates a linear history rather than a DAG.
Given a commit history,

```
        A---B---C topic
       /
   D---E---A'---F master
```

Running `git rebase master` will produce

```
                 B'---C' topic
                /
   D---E---A'---F master
```

git bisect

`git bisect` runs a binary search to find the first bad version.

```
git bisect start
git bisect bad (current version)
git bisect good VERSION
```

Note: `git bisect` might not work on merged commit histories.

15.5 Extraneous git Features

15.5.1 Tags

Tags essentially label commits, and are created by running the command `git tag COMMIT_ID`. There are various types of tags: plain, annotated, and signed tags. They are located in `refs/tags`. It is worth noting that branches and tags can be the same names.

Plain

Plain tags are the literally just giving names to commits. There is no metadata stored.

Annotated

Annotated tags store metadata and can be created by running the command `git tag -a TAGNAME -m "MESSAGE" COMMIT_ID`.

Signed

Signed tags are for security, and have cryptographic authentication. They can be created by running the command `git tag -s`

15.5.2 Submodules

Submodules in git are used to "point" to another project. It contains the commit ID's within the other project and is used for version stability. To update submodules, run the command `git submodule foreach git pull origin master`.

15.5.3 Stashing

Stashes are implemented with a stack, and are used for switching branches. `git stash push/pop` will push/pop your modified working files onto/off the stack respectively. `git stash list` will list all of your stacks. If you want to be avant garde like Eggert, you can instead do:

```
git diff > mychanges.diff
patch -p1 < mychanges.diff
```

15.6 Communicating Between Developers

There are multiple ways to communicate between developers:

GUI enthusiasts: Share a repository and use pull requests via something like Github

CLI enjoyers: Email patches back and forth:

`git format-patch A..B`

`git send-email`

`git am FILE` (automatic merge)

16 Build Tools

Who is the audience for these build tools?

Developers: Write the source code for the software

Builders: Compile source code for a particular platform

Distributers: Ship programs to users in the form of distributions

Installers/Configurers: Users that install and use the programs

16.1 `make`

Once developers are done writing their code, they want to help builders compile it. In order to do so, they must provide metainformation about the source code and all of its dependencies. One easy way to do this is by writing a metaprogram that will automatically implement these build instructions. In simple programs, rather than a metaprogram, a README is used. Otherwise, we can write a simple script (commonly labeled `build.sh` or `setup.py`). Here's what a sample script may look lke:

```
gcc -c a.c
gcc -c b.c
gcc -c c.c
gcc -c a.o b.o c.o -o foo
```

16.1.1 Flaws/Fixes

There are a couple downsides to this approach

(a) Maintaining this file can be too time consuming/get confusing

(b) Rebuilding after small changes is expensive

(c) Not scalable

(d) It's slow (missing parallelism)

How do we fix these issues? We can't fix all of these issues, but we can use a separate build tool rather than write our own script via `Makefiles`.

16.1.2 Makefiles

Makefiles are similar to shellscripts but are more efficient: they only rebuild what is necessary. A sample Makefile may look like:

```
a.o: a.c
    gcc -c a.c
b.o: b.c
    gcc -c b.c
c.o: c.c
    gcc -c c.c
foo: a.o b.o c.o
    gcc a.o b.o c.o -o foo
```

`make` will determine what needs to be rebuilt by looking at file timestamps. Additionally, we can run jobs in parallel with `make -j 10`This solves (a), but this approach also creates new problems/doesn't fix old problems.

(i) Clock skew: Different machines might differ in their exact system time and if they're operating on the same set of files, it's possible that one system writes a timestamp that is ahead of another system's time or the program file generated by make is older than the edited timestamp.

(ii) Missing/Extra Dependencies may cause the program to break//rebuild unnecessarily.

16.2 Syntax

16.2.1 $

`$`: Expands a variable

```
OBJ = a.o b.o c.o
foo: $(OBJ)
    gcc $(OBJ) -o foo
```

is equivalent to

```
foo: a.o b.o c.o
    gcc a.o b.o c.o -o foo
```

16.2.2 $@

`$@`: Expands to the rule name.

```
foo: a.o b.o c.o
    gcc a.o b.o c.o -o $@
```

is equivalent to

```
foo: a.o b.o c.o
    gcc a.o b.o c.o -o foo
```

16.2.3 Rules and Recipes

Rules have the following syntax:

```
TARGET: DEPENDENCIES
    RECIPE
```

Note: Recipes are shellscripts. Furthermore, `make` is a thin layer around the shell.

17 C (The Superior Language)

C is the predecessor to C++, so it is missing a lot of 'features' that C++ has. Some of these are:

(a) STL

(b) Classes and Objects

  (i) Polymorphism (`foo(int& a)` and `foo(bool a)`)
  (ii) Inheritance (`class Dog: public Animal`)
  (iii) Encapsulation (`private`)

(c) Namespace Control

(d) Explicit use of `static` to create singular instances

(e) Exception Handling

(f) Memory Management: `new` and `delete` (wrappers for `malloc()` and `free()` respectively)

(g) `cin`, `cout`, `<< >>`

(h) Function Overloading

17.1 Architecture of a C Environment

Compilation is broken up into different stages:

(1) Preprocessing (`gcc -E foo.c` $\implies$ `foo.i`)

(2) Conversion to ASM (`gcc -S foo.i` $\implies$ `foo.s`)

(3) Create Object Files (`gcc -c foo.s` $\implies$ `foo.o`)

(4) Linking (`gcc *.o` $\implies$ `a.out`)

Note: At (3), the object files have holes in them. We need to resolve this by linking all of the `.o` files which produces a single executable which will cut and paste all of the `.o` files in the correct place.
Note: The preprocessing phase is usually omitted by higher level languages (e.g. Python). Essentially, preprocessing allows for conditional compilation via `#ifdef, #ifndef, #endif`, and other macros.

18 Debugging

Debugging a program serves two main purposes:

(1) Correctness: Verifying that the expected output matches the actual output

(2) Performance: Change code to optimize for hardware/better performance

In real-time systems (car brakes), correctness and performance are indistinguishable, since they are dependent on each other. In general, try to avoid using a debugger (Eggert's words not mine). Below are some alternatives to debugging that should be tried before busting out a debugger.

(1) Print Statements (`cout`, `printf()`, ...): To track variable states

(2) `time`: To measure the efficiency of the program (and deduce any timing issues)

(3) `ps -ef`: Prints all active processes

(4) `ps -efjt`: Similar to `-ef`, but in tree form

(5) `top`: List of top-consuming processes (by CPU %)

(6) `kill`: Kills a process

(7) `strace ./a.out foo`: Logs to `stderr` all system calls

Note: Most often, (1) and (2) are most commonly labeled under developer tools, while (3) and its subitems are labeled under operation team tools.

ASIDE: System Calls

System calls are special commands executed by the OS kernel, which lives right atop the hardware level. This is more of an OS topic but it still proves relevant in this course, especially since we talk about the `gcc` compiler. System calls are special since only the OS kernal can actually execute these instructions. Most other applications must **ask** the OS kernel to execute the syscall.

18.1 `valgrind`

`valgrind` is a debugging tool mainly used to detect memory-related bugs and to log **all** instructions a program executes.
`valigrind ./a.out foo`
`valgrind` isn't perfect, but it does help against many trivial memory-related bugs such as bad references. `valgrind` will catch

```
char *p = NULL;
*p = 'x';
```

but won't catch

```
char a [10000];
char *p = &a[10000];
*p = 'x';
```

since `valgrind` won't do trivial boundary checks by default. **See HW 5** for more information.

ASIDE: The Stack

`gcc -fstack-protector` is there for a reason: to prevent malicious people from injecting code into the program's instruction list, overflowing the buffer, and taking control.

18.2 `gcc`

`gcc [OPTIONS] [FILE]` has many options to help you debug. Here is an important one:
`-fstack-protector`: Protects against stack overflow errors by inserting a canary right around stack boundaries. If the canary is not a predictable value, the stack was corrupted, so the program will crash gracefully. Note that this won't always work since there are ways to get around this and still cause stack overflow errors.

18.2.1 Profiling

`gcc --coverage` will profile your program, creating a temperature graph by injecting code into your program like

```
if(x < 0)
   counter[19246]++;
   f();
```

and will output `counter` to an output file (`counter` is the profile). Note that profiling is input-dependent.

Profiling is done to find bugs with cold functions (a.k.a why are the cold?). However, this is also test-case dependent, since if functions are labeled cold, it might be because your test cases never touch them.

18.2.2 Static Checking

Static checking prevents your code from compiling if it fails a static check/assert and are used to document your code and assumptions. They have the format
`static_assert(E)`, where `E` is a constant expression. So, asserts like

```
int f(int n) {
    static_assert(0 < n);
}
```

will not work, since `n` is not a constant variable.

18.2.3 Warning Flags

`-Wall`: `gcc` will turn on all "useful" warning flags

`-Wcomment`: Catches bad comments like `/* bad /* comment */`

`-Wparentheses`: Catches potential arithmetic errors like `return a << b + c` (`+` has higher operator precedence than `<<`)

`-Waddress`: Warns about using addresses that are probably wrong. e.g. Consider the following:

```
char* p = f(x);
if (p == "abc")
    return 27;
```

`-Wstrict-aliasing`: Warns against "bad" casts. e.g. Consider the following:

```
long l = -27;
int *p = (int*)&l;
*p = 0;
```

`-Wmaybe-uninitialized`: Warns if you're using potentially uninitialized variables.

18.2.4 Optimization

`gcc` has an optimization flag that will trade compile time for faster executables.
`gcc -O#` (0-4, 2 being the most common) will determine the level of optimization.

18.2.5 Overview

The two most common ways `gcc` optimizes your source code is by caching in registers and executing out of order. This makes your code harder to debug when you run it, since what you see is not always what you wrote in the source code.

18.2.6 `-O#` Alternative: `-flto`

`gcc -flto`: An alternative to the plain `-O#` flag, we have `gcc -flto`, or File Time Link Optimization. This will put a copy of the source code into all of the `.o` files and will optimize the entire program at once, with all of the modules linked. This way, there is more opportunity for optimization. The main downside to this approach is that compile times are even slower.

18.2.7 Built-In Compiler Functions

Below are a list of common functions to help optimize or debug your source code:

(a) `__builtin.unreachable()`: Tells the compiler that if the program ever reaches `unreachable()`, then behavior is undefined. This allows for further optimizations. e.g. Consider the following:

```
if(x < 0)
    __builtin.unreachable();
return x / 16;
```

Since the compiler knows that `x` *should* never be negative, it can use the bitshift operation `x >> 4` to optimize.

(b) `__attribute__(ATTR)`: Advice to the compiler (can be ignored). Does not change the program. This allows for further, nuanced optimization. e.g. Consider the following:

```
#ifdef __GNUC__
#define __attribute__(x)
#endif
```

The above code will disable the attribute if compiled with a non-gcc compiler.

18.2.8 Attributes

```
charbuf[1000]__attribute__((aligned(8)));
```

`aligned(x)` makes sure that `charbuf` has an address with a multiple of `x`, where `x` is a power of 2. This is to maximize the number of CPU cache hits. Since RAM is divided into cache boundaries, the machine will cache (usually) 64-bytes of memory on the CPU. Doing `aligned(x)` will (try to) ensure that the array fits into the cache's 64-byte boundaries.

```
void func(void) __attribute__((cold))
```

`(cold)/(hot)` labels a function either cold or hot, respectively. A cold function is one that is rarely executed, whereas a hot function is one that is executed frequently. The motive behind this is so that the instruction pointer does not have to jump around everywhere and can execute (relatively) sequentially.

```
instruction pointer
V
------------------------
| hot | program | cold |
------------------------
```

This is how the compiler will order your code using attributes.

```
int hash(char*, ptrdiff_t) __attribute__((pure, access(read_only, 1)));
int a = hash(p, 27);
int b = hash(p, 27);
```

`pure` means that there is no user-visible storage. In this case, `a` must equal `b`.

```
int square(int)__attribute__((const));
```

`const` means that the value is both `pure` and does not depend on user-visible storage. In C, `pure` ≡ `[[reproduceable]]` and `const` ≡ `[[unsequenced]]`

```
void *myalloc(ptrdiff_t) __attribute__((alloc_size(1), malloc(free, 1), returns_nonnull))
```

18.2.9 Runtime Checking

`-fsanitize=undefined`: Runtime check for overflows

`-fsanitize=address`: Crash if bad pointers are used

`-fsanitize=leak`: Check for memory leaks

`-fsanitize=thread`: Check for race conditions

ASIDE: `unsigned`

`unsigned` is a disaster for one very specific reason. Let `x` be an unsigned integer. Now consider:

```
if (x <= -1)
```

This statement will always evaluate to true because `x` is unsigned. Logically however, this makes no sense.

18.3 Debugging: Using gdb

There are a couple prerequisites before using gdb:

(1) Stabalize the failure (make sure it consistently breaks)

(2) Locate the source of failure (point of failure)

(3) Optionally, `gcc -g` will put information such as names of local variables to make debugging easier.

```
1. (gdb) set cwd /usr
2. (gdb) set env TZ American/Chicago
3. (gdb) set disable randomization on(default)/off
4. (gdb) r -c foo < bar >baz
5. (gdb) r
```

(gdb) `attach PID`: Takes over process id

(gdb) `b foo`: Breakpoint at `foo`

(gdb) `info break`: Lists breakpoints

(gdb) `del #`: Delete breakpoint `#`

(gdb) `step, s`: Step to the next line

(gdb) `stepi`: Step into the next machine instruction

(gdb) `next, n`: Step over function calls

(gdb) `cont, c`: Continue execution

(gdb) `fin`: Finish current function

(gdb) `bt`: Backtrace (examine current state)

(gdb) `p E`: Print the value of the expression `E`

(gdb) `target TARGET`: Target a specified architecture

(gdb) `reverse continue, rc`: Reverse execution

(gdb) `checkpoint`: Will output a unique id of the program state

(gdb) `restart ID`: Restarts execution starting from `ID`

(gdb) `watch E`: Pause execution when `E` changes

18.3.1 gdb with Optimization

When debugging it is important to remember that the executable may behave differently than what is written in the source code due to **optimization (See Optimization)**.

Out-of-Order Execution

Consider the following source code:

```
(1) q = a / b;
(2) r = a % b;
```

may turn into

```
r = a % b;
q = a / b;
```

since, in a lot of architectures, the instruction `idivq` will calculate both the division and modulo. This is due to the "as-if" rule: The compiler can generate any code whose behavior is "as if" it did the obvious. Therefore, one method of debugging is to do the following:

```
gcc foo.c
gdb a.out
[debug]
gcc -O2 foo.c
[run]
```

The problem with this is that the optimizer may be buggy (unlikely), or the optimizer exposed a bug that wasn't caught when debugging (more likely).

18.3.2 Finding Bugs

Suppose

```
start
...
bug triggered*
...
...
failure
```

How do you find the point of failure(`*`)?
In small programs or programs with easy test cases, we can:

Come up with a reproducable test case

Make sure the program doesn't take too long to execute

Rerun the program until you find it

For larger programs, we can use gdb's **Reverse Execution** to find the bug(s).

Reverse Execution

gdb will start executing the program backwards. Note that this is a very expensive process since gdb has to cache all program states. To efficiently use `gdb rc`, we can use commands like `checkpoint`, `restart` and `watch`. Catchpoints stop the program if it throws an error (similar to a try/catch block). ***Note:*** `gdb watch` is so cool that a lot of architectures have hardware support for `watch`, meaning it's fast. On x86-64, you can `watch` up to 4 memory locations.

18.3.3 Review

Try not to do it; that is, write good test cases

Test cases > source code: Test-Driven Development - Write test cases before coding the corresponding part

Use a better platform: e.g. Subscript errors? C++ $\rightarrow$ Rust or Java

Defensive Programming

Assume other devs are useless

Runtime checking

Trace/log what the program does along the way (helps debugging later)

Assertions

Exception handlers (try/catch)

Barricades: Middleware to take in any data and only pass through "safe" data into the program

19 `git` Internals

19.1 Preface: Atomicity and SHA-1

An atomic operation only has two states: not executed or executed e.g. `cd`. Non-atomic operations such as `cp` are logical since it is possible to be in the middle of writing a file when execution stops (unexpectedly). `git` uses many atomic operations to keep the working tree clean and to prevent corrupting the repository. For example, `git commit` is built atop atomic operations because it would not be good to only have half of a commit.

The SHA-1 checksum is the hash function that `git` uses to create commit ids and object hashes. Though it has been cracked, `git` still uses it because:

> The probability of collisions is $\frac{1}{2^n}$, where n is 160 in this case
>
> Finding a byestring to match a given hash is expensive ($O(2^n)$) (SHA-1 is a one-way hash)
>
> Finding collisions is expensive ($O(2^n)$)

19.2 Overview

`git` is like an application-specific "file system" (because it was built by file system designers). It is built atop an ordinary file system and has many similar issues that file systems have. `git` is split up into to parts: plumbing and porcelain. The plumbing part deals with the internals such as data structures and low level commands, while the porcelain part is what the user interfaces with (e.g. `git commit`).One of the main issues with `git` is distinguishing data with metadata.

19.3 .git/

Below are some of the subdirectories/files under `.git/` and their usage.

`.git/ branches/`: Legacy folder (for backwards compatability) that used to store branches

`.git/ config`: `git`'s configuration file. Analogous to a barricade **(See Debugging)**

`.git/ description`: Descriptor for the repository

`.git/ HEAD`: The pointer[4] that points to the tip of the current working branch

`.git/ hooks/`: `git`'s callbacks

`.git/ index`: Binary data structure keeping track of your commit

`.git/ info/exclude`: Contains blobs that git will ignore (like `.gitignore`, but not for working files)

`.git/ logs/`: Record your branch tips and logs changes to branches (2^{nd} order history: history of the **repository**)

`.git/ objects/`: Contains the object database with records of all objects managed by the repository

`.git/ refs/`: The references directory to store commits and tags

`.git/ packed-refs`: Condensed version of `refs`

Below are some more notes on the following contents of `.git/`

config

There is no mixing of data and metadata. That is, do not include anything in the .git folder in the repository. This leads to a very natural question: How do I share my `.git/config` file? The solution is to write a script to set up the `config` file and put instructions in a README.

[4] See *Pointer* in **Managing the Repository: Branch Manipulation**

objects/

Objects in `git` are identified via a 40-digit hexadecimal (160-bit) checksum[5]. The objects folder will store all objects managed by `git`. The subdirectories (e.g. `objects/0f`) contains the first two digits, while the file descriptor inside contains the remaining 38 digits. This was done to meet the storage requirements at the time. Nowadays, it's still formatted this way for backwards compatability.

refs/

`refs/heads/BRANCH_NAME`
Points to the last **local** commit ID of `BRANCH_NAME`. e.g. `.../main` will contain the most recent **local** commit ID of `main`.

`./remotes/origin/HEAD`
Contains the relative file path of where HEAD points. e.g. `ref: refs/remotes/origin/main`.

`refs/remotes/origin/BRANCH_NAME`
Points to the last **remote** commit ID of `BRANCH_NAME`. e.g. `.../main` will contain the most recent **remote** commit ID of `main`.

`refs/tags/TAG`
Contains all of the repository's tags

19.4 Representing Objects in `git`

Objects in `git` are not files. Rather, they are a blob containing a hashed byte-string. The following subsections will manually build a `commit` object.

19.4.1 Working Files → `blob`

The command `git hash-object FILENAME -w` will create an object with the 40-digit SHA-1 checksum (hash) for its file descriptor. Note that if two files have the exact same contents, then `git hash-object` will return the same 40-digit checksum.
Note: `git cat-file -p/t HASH` will print either the contents or type of the object with the hash `HASH` respectively.

19.4.2 `blob` → `tree`

The command `git update-index --add --cacheinfo <MODE> <HASH> <FILENAME>` will add the object to the index, where `MODE` is the type of object (e.g. `blob` = 100644). The first 3 digits is the filetype (100 = regular file) while the next 3 is the octal representation of permissions (644 = o+rw, ag+r).
The command `git write-tree` will create a tree object using the current index.

19.4.3 The commit Object

A `commit` object contains the following:

tree

commit message

author + timestamp

committer + timestamp

parent commit(s)

Note: `BRANCH_NAME` is a commit object. More generally, branches deal with commit objects. Additionally, `git` compresses objects.

[5]The checksum is calculated via the SHA-1 hash, and is used to avoid collisions.

19.5 Compression

19.5.1 Overview

Compression is the process of reducing file size while preserving as much data as possible. Many techniques are used to compress data, and the various compression algorithms are application-specific. There are trade offs to compressing: CPU time to compress/decompress, % compressed/decompressed, and RAM usage are all inversely related.

Problems
If any data gets corrupted during compression or decompression, neither algorithms works and any remaining data is now suspect.

19.5.2 Huffman Coding

The algorithm for huffman coding is very straightforward:

(1) Sort character frequency in non-decreasing order

(2) Take the least two likely symbols with the smallest weights and combine them, adding their weights

(3) Delete the two individual symbols from the list and add the new combined symbol(s) to the list

(4) Repeat (2) and (3) until there is only one node left

Adaptive Huffman Coding is a variation of the huffman tree, in which the decompressor builds the Huffman tree as it receives data, updating the tree in real-time.

19.5.3 Dictionary Compression

The Dictionary Compression algorithm is similar to a sliding window algorithm, and is as follows:

(1) Create a dictionary of byte string

(2) Send one byte string at a time, sending the offset and size between a recurrence (if there is one) and the first occurrence (if within the sliding window) instead.

(3) Repeat until End of File

19.5.4 `git` Compression

To compress objects, `git` uses `zlib/gzip` which use both **Huffman Coding** and **Dictionary Compression** (e.g. Raw Data → Dictionary Compression → Huffman Coding).

20 A 1h 20m Aside: Character Encodings

20.1 Overview

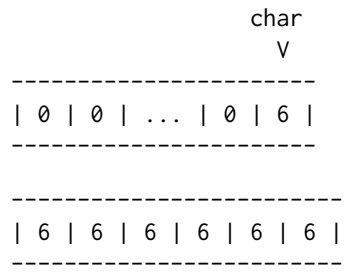
In computers, there is no such thing as a character. Computers only store numbers, so characters are just mapped integers. An easy example is the C/++ character. In C/++, the character 'x' can be represented as 'x', 120, or '\170'. Therefore, characters are just an individual symbol that corresponds to a small integer.

Corollary

A character string is a sequence of characters. From above, we have that a character is just an integer. So, it follows that a character string is a sequence of integers.

20.2 Dark Ages

In 1960, There were only 64-bit character encodings: A-Z, 0-9, +, -, *, /, etc. There is a problem with this approach however. If, by example, the wordsize is only 24 bits, 26 bits are being wasted. A simple fix is to afix wordsizes to be 36 bits. Then, take a corresponding 36-bit word and divide it into 6 blocks, where each block is any character that can be represented with 6-bits. Below are diagrams for the 24-bit and 36-bit wordsizes respectively.



20.3 EBCDIC

In 1964, IBM System 360 (Mainframe) introduced byte addressing which separates addresses of bytes. They used 8-bit bytes and 32-bit/4-byte words. Current x86-64 machines have 512-bit registers and 64-bit words. EBCDIC expanded the character set to 8-bits.

20.3.1 Flaws/Fixes

For some reason, they did not make lowercase letters contiguous and left gaps/holes in the character encoding table. These idiots did not listen to Eggert and clearly did not follow test-driven development, since they would've made it better otherwise. This is why no one uses it anymore.

There are no fixes for this bum-ass character set. Notably, Eggert wasn't able to write a C program that did character arithmetic, so they got an F in CS35L and did not pass.

20.4 ASCII

ASCII is a 7-bit character set and is superior to EBCDIC since they listened to Eggert's request of wanting to write a C program that did character arithmetic. They use 8-bit word sizes, but the first bit is a parity bit[6]. There are 32 control characters that won't print to the console (0-31st characters on the table). Some interesting things to note is that NULL is all bits 0 (7'b0) and DEL is all bits 1 (7'b1) for historical reasons.

20.4.1 Flaws/Fixes

ASCII does not natively support other languages since it's character set is so small. The devs clacked their three braincells together and came up with ISO/IEC 8859 (why 8859 I have no idea), which was a guide for how to extend[7] ASCII to other languages.

8859- 1: Latin-1 (Western-European languages)

8859- 2: Latin-2 (Central + East-European languages)

8859- 3: Latin-3 (Southern-European languages)

8859- 4: Latin-4 (Northern-European languages)

8859- 5: Latin-5 (Cyrillic languages)

8859- 15: Latin-9 ("Fixed" Latin-1 which added some bullshit French character and minor languages, and added the euro symbol)

[6] A parity bit XORS all of the other bits for error detection

[7] These extensions were not allowed to collide with the original ASCII encodings

While these were great bandaids, these bums clearly fell asleep in Eggert's lecture on test-driven development, since these extensions are not cross-compatible. Furthermore, metadata for character encodings is required to determine which character set to use when parsing (e.g. For HTTPS, we have Content type ... charset = "ISO 8859-1" in the header). Lastly, the developers did not take into consideration Asian languages, which I can't really knock them for since Asian languages have character sets longer than my notes for this class.

20.5 Encoding for Asian Languages

Developers said "fuck it we ball" and increased to 16-bit character sets to encapsulate Asian languages like basic Chinese. In C, we cannot use `char` anymore, so we have to use `short`'s.

20.5.1 Flaws/Fixes

The problem now is that it's completely incompatible with any other character encoding schema. e.g. Something like "Hello" will be parsed (in ASCII) as

```
-------------------------------------
| 0 , 'H' | 0 , 'e' | ... | 0 , 'o' |
-------------------------------------
```

where `0` is the null-byte. Furthermore, this encoding is very obviously bloated.

To fix the incompatibility, they used multibyte characters, which had the following format:

1-byte characters for ASCII had parity bit 0

2-byte characters for others (e.g. Kanji) had parity bit 1

This encoding was called ShiftJIS and was adopted by Microsoft and ASCII[8]. These developers were big fans of the Hydra[9] because their "fix" also introduced two issues. Firstly, the file **must** be processed sequentially due to character **context**. Moreover, this schema introduced more invalid encodings.

20.6 Unicode Consortium

Unicode was an attempt to "unify" Asian languages and have a single universal charcter set for all characters and languages. There are currently 149,186 assignments. In the 1990's, the developers did not futureproof for emojis and though that a 16-bit character set would be enough.

20.6.1 Flaws

Unicode has a lot of repeat characters that are virtually identical but there were national debates over some goddamn lines and that's why we have a lot of repeat characters (most common in Asian languages). One of Eggert's favorite examples is the Latin vs. Cyrillic 'o'. They look the same but apparently there's a slight difference. I'm not going to do a `diff` of the character pixel maps so I'll take his word for it.

20.7 UTF-8

UTF-8 is upwards compatible with ASCII. Its schema is as follows:

Every multibyte sequence has only non-ASCII bytes (parity bit 1). This way, it is easy to see character boundaries

There are 3 byte types:

ASCII byte: parity bit 1

Continuation byte: parity bit 1 and 2nd bit 0. It an **never** be a leading byte.

Length + Leading bits byte: First k bytes are the length of the character

[8] ASCII was a Japanese company completely unrelated to US-ASCII (similar to how Javascript is not related to Java in any way)

[9] In Greek mythology, the Hydra was a serpentine water beast which, when one of its heads were cut off, two more would grow back in its place

UTF-8 Boundaries

```
------------
| 0XXXXXXX |
------------
```

U+0000 - U+007F

```
------------  ------------
| 110XXXXX |  | 10YYYYYY |
------------  ------------
```

U+0080 - U+07FF

```
------------  ------------  ------------
| 1110XXXX |  | 10YYYYYY |  | 10ZZZZZZ |
------------  ------------  ------------
```

U+0800 - U+FFFF

```
------------  ------------  ------------  ------------
| 11110XXX |  | 10YYYYYY |  | 10ZZZZZZ |  | 10WWWWWW |
------------  ------------  ------------  ------------
```

U+FFFF - U+10FFF

20.7.1 Flaws

No character encoding is perfect, UTF-8 included. There are gaps in UTF-8 encoding since there are multiple ways to spell characters:

`11000001 10111111`

is technically the DEL key, but these encodings were accounted for (as invalid UTF-8 encodings) since the developers did not fall asleep in Eggert's lecture on character encodings. Moreover, byte-for-byte comparisons won't work because something like `strcmp("UCLA", "UCLA");`, where the first and second UCLA's are 1-byte and 2-byte respectively, will return false. Additionally, something like

```
char *p = XXXXXX;
p[strlen(p)/2] = 0;
```

won't work in UTF-8.

More invalid UTF-8

```
| -------------
| | 10XXXXXXX |
| -------------
```

Continuation byte **must** follow length bytes

```
------------
| 111110XX |
------------
```

Max length is 4

```
------------ |
| 1110XXXX | |
------------ |
```

Length bytes must be at the start*

Note: This may be a part of a datastream that hasn't sent all of its packages over yet, so you have to be careful when checking for valid UTF-8 encoding. This is why `Barricades` are important.

One common coding convention is to use ASCII only to prevent any encoding errors.

21 Backups

According to Eggert, we backed up a total of 100 ZB[10] in the past year, roughly 90% of which is dulicate data and roughly 50% in the cloud. Backups very clearly dominate storage, and there is a cost for backups (global warming, apparently). Do we need all of these backups? If you look at M152A computers, you'll know that to a certain group, 93 backups (with extremely similar names) are necessary for a singular lab.

21.1 Overview

Backups are a snapshot of file contents (with metadata for each file). There are two types of backups: abstract and concrete.

Abstract: Each file is a byte string (byte sequence with separate byte strings for data, metadata, etc.). This means it's dependent on OS but it isn't wasteful since you only copy over exactly what you need.
Concrete: Abstract the actual data into blobs[11] and instead, copy the blocks in the underlying device. This means it's independent of the OS and captures the exact state of the device, but it could potentially be wasteful since in practice, the device might contain bloat.

Regardless of methodology, backups address a multitude of problems:

(1) Data loss

(2) Hardware failure

(3) Tracking history

(4) Accidentally trashing a working copy because you didn't follow Eggert's Best Practices™

(5) Corrupted drives (Hardware failure but with some chest hair)

(6) Security (ransomware)

Backups used to just be an operaton staff (Ops) problem, but they couldn't handle it so now it's a DevOps problem.

21.2 Cheaper Alternatives

(1) Simply generate less data, use compression or back up less often (who would've thought)

(2) Multiplex your backup: multiple drives backed up onto one bigger drive

(3) Incremental backups: Back up only what changes. Note that this is more fragile (but is very similar to (1))

(4) Selective backups: Determine what is worth backing up.

(5) Snapshots: **See Snapshots)**

(6) Backup to cheapder devices. e.g.

```
Flash => Disk => Optical
Main    Backup  Secondary
                 Backup
```

(7) Redundancy in devices: **(See RAID (Redundant Array of Inexpensive Disks))**

[10] 1 ZB = 10^{21}

[11] Blobs stand for "Binary Large OBjectS" and "isn't made up", which I don't really believe but whatever.

21.2.1 Incremental Backups

At the file level, each backup has a timestamp, so take a similar approach to `make` **(See `make`)** and only backup files with t' > t. Consequently, we run into the same problems as `make` like clock-skew. So, in yet another layer of abstraction, we rely on the clocks being monotonically nondecreasing. One other problem with this is that deleted files are not addressed in this schema.

Within a given file F, consider ΔF. You can do `diff -u F ΔF > t`. You now have an "edit script" that will patch a file F to ΔF by running `patch <t F`. This is good for text files.

21.2.2 Automated Data Grooming

Deduplication is the process of automatically removing data we don't need. The algorithm works as follows:

```
find all file where g == f
    for each g
        rm g
        ln f g (there is a race condition BUT ln -f f g is atomic)
```

This assumes the files are read-only, since if you now change `g`, `f` is also changed **(See Hard Links)**. To remedy this problem, we have Copy-on-Write (CoW), which will make a copy of a file if it's link count > 1, writing to the copy. The idea is to share read-only files, and make a copy for writes.

This leads to another issue: If metadata(f) ≠ metadata(g), g will lose metadata. To solve this issue, we change the definition of equality. Finally, there's the issue of not having enough storage to copy on write.

Block-level Deduplication

Let a particular file system have 8 KiB blocks. We can represent it as:

```
-----------------
|   | A  |   | A |
-----------------
```

Using block-level deduplication, there is only one copy of A. More generally, the file system will only save distinct blocks (this is default on many Linux distros). This way, we get an implicit Copy-on-Write for free. There are three main issues with this type of deduplication:

(1) Allocation: Not enough storage to copy on write

(2) Slower access time: "What's another level of indirection?" is what the devs said, laughing

(3) Reliability: If a block goes bad, you're screwed

21.3 Backups and Encryption

Reasons for encrypting backups:

(1) You don't trust your cloud provider

(2) You don't trust your operations staff (lol)

(3) Data must be encrypted for other reasons (security)

21.4 Bridge to Version Control Systems

21.4.1 Preface: Versioning and File Systems

Do applications need to know about backups?

Yes: Software like Files-11 (OpenVMS) will create viewable backup files, so when you do `ls -l`, you get something like

```
foo.c; 1
foo.c; 2
...
```

so that applications now have an API for versioning.

21.4.2 Snapshots

No: Utilize snapshots, which captures the current state of your file system in user-specified invervals. This method is used on SEASnet via a NetApp file server that runs WAFL (block-level deduplicaiton).

ASIDE

Directory size is irrelevant (it has a nice personality). Why? You can't directly read from a directory; that is, you cannot do something like `cat DIRECTORY`.

21.4.3 History

SCCS in 1972 was the first major proprietary VCS, and it worked as follows: for each source file F, $\exists$ s.F which contained the entire history of F in increasing time order as well as metadata (committer, message, etc.). This let a user read any version via a single sequential pass. However, the downside was that the cost of retrieval, at worst, was now O(size of history).

A free alternative was RCS, which was similar to SCCS, but structuerd as follows: for each working file F, $\exists$ RCS/F.v, where F.v was the history of the file in the format:

```
------------------
     Metadata
------------------
Most recent change
------------------
Reverse time order (e.g. 12 => 11)
------------------
...
------------------
2 => 1
------------------
```

One major issue with RCS was that it was a per-file VCS. The creator of RCS wasn't as smart as Linus Torvalds.

CVS (not the pharmacy) introduced commits that can address multiple files, and had a client-server model for repositories. A descendent of CVS was SVN, which was CVS on steroids.

The Linux kernal initially used: CVS $\rightarrow$ SVN $\rightarrow$ BitKeeper (proprietary software). Linus Torvalds said "fuck that I want free" so naturally, he built `git`, which hilariously ran BitKeeper out of business (they open sourced in 2016 but hardly anyone uses BitKeeper anymore).

22 A 10 min Overview of Compiler Internals

Compiled languages (like C/++) compile in multiple stages **(See C)**. The hardest part however is converting into general ASM. Compilers answer the question of "How do I turn
`a += *b[5]` into
`movq b, %rbx`
`movl 0 XX`"

Let L denote the many source languages (C/++, Python, etc.) and M denote the many architectures (x86-64, ARM, RISC-V, etc.). Do we have to write $L \cdot M$ compilers? No! Instead, we have a set of common compiler internals that take in a language $l \in L$ and translate it to a specific architecture $m \in M$, which then converts into general ASM. This way, we only need to do $c + L + M$ work, where c is a constant.

23 Software and Law

23.1 Software

Software is:

A set of instructions to a computer

A way to collaborate with other users and developers to solve problems

23.2 Law

Law is "the art of predicting judges"[12]. It can be broken up into multiple categories:

How to collaborate

How to deal with failures in collaboration

Civil/Contract/Commercial

Criminal

Constitutional

International

Admiralty (oceanic)

23.2.1 Commercial Law and Software

Back in the Dark Ages, copyrights and patents were very different. Copyrights were reserved for creative works like books, while patents were reserved for functional inventions like a urinal headrest[13]. Nowadays, the line between copyrights and patents are starting to blur due to software.

Software is used with hardware, but software would technically be copyrighted while hardware would be patented.

Trade Secrets

Trade Secrets have no expiration date, and expire when the secret is disclosed. Note that if you illegally disclose a secret, it is still legally a secret. There are agreements to keep secrets called "Trade Secret Agreements". This is more commonly referred to as an NDA, or a Non-disclosure Agreement and many companies make you sign one.

[12]This quote was authored by Paul Eggert, UCLA Senior Lecturer, in La Kretz Hall 110 on February 16, 2023
[13]Hilariously enough, this was a real, granted patent.

Trademarks

Like Trade Secrets, Trademarks don't expire until a company stops using it. The goal is to avoid customer confusion. So, if trademarks don't collide, it's ok (e.g. Apple computers and Apple Records).

Personal Data

Whenever you visit a site, websites have access to your IP address and browser fingerprint.

Copyright

Copyrights cover creative works, and protects the form, not the idea. (e.g. I can write a book about whale-catching and I wouldn't be infringing on Moby Dick's copyright). Inversely, the Public Domain is any creative work that is free to use and isn't copyrighted.

Patents

Patents cover practical works like inventions and utility. To be granted a patent, you have to apply for one, and it gets reviewed. The invention must be: novel, useful, and it has to work.

23.2.2 Infringement

Legal protection for copyright/patent holders (under civil law). Infringement penalties include damages (actual[14] or statutory[15]) and takedown notices (DMCA).

23.2.3 Technical Protection

For software, you can use SaaS (Software as a Service) or program obfuscation.

23.3 Licensing

A license is **not** a contract, but rather a grant permitting you to do something, and is often part of a contract and has strings attached. When do they come up?

Buy vs Build: Using already developed software or writing your own

Derivative works: Building off of other people's work

The different types of licenses (free → proprietary) is as follows:

Public Domain: Free use

Academic: Must give credit (e.g. MIT License)

Reciprocal: Share and share alike (e.g. GNU Public License)

Corporate: e.g. Apple, Oracle

Proprietary: Paid service

23.3.1 Dual Licenses

Products can be distributed under multiple licenses (e.g. MariaDB has a proprietary version and a free (1 yr delayed) version). The reasoning for licensing and free software is so that you are in the company's ecosystem.

23.4 Software and Laws of War

"casus belli" and "jus ad bellum" translate to "case for war" and "justification for war" respectively. Is a software attack enough justification to go to war?

[14] Actual: Calculated losses
[15] A minimum they pull out of their ass