GENERAL

Distributed Computing

- Field of CS studying the **distributed system**, a system whose components are located on different **networked computers**, which then communicate and coordinate their actions by **message passing** to each other.
- Ex: MMO games

Context

- Minimal set of data used by a **task** (ex: process, thread)that must be saved to allow a task to be **interrupted** and later continued at the same point
- The smaller the context is, the lesser the **latency**

API

API

- Application programming interface is a set of subroutine definitions, protocols, and tools for building application software.
- APIs lets software communicate with each other

REST

- Representational State Transfer (REST) is an architectural style that **defines a set of constraints and properties based on HTTP**
- A **web service** that conforms to REST architectural style is a **RESTful web service**
    - This allows interoperability between computer systems on the internet.

REST API

- An **API that uses HTTP as a protocol**.
    - Ex: An application uses an HTTP GET request to get data from the Google Maps API. Or the app uses a HTTP POST request to write data (and also get data in response) to the Google Maps API. This API is a REST API, as it used HTTP.
- The API is the **server component** in which REST requests are addressed. These requests take the form of: HTTP GET, POST, PUT, POST, DELETE.
- Requests to modify the database (that the server handles) are authorized using **OAuth.**
    - Read requests do not require authorization except for user details.

POSIX File Access API (Linux, C)

- **File Descriptor**
    - A file descriptor is an int
    - When a process starts, it normally has three already open file descriptors:

        - 0: standard input (read only)
        - 1: standard output (write only)
        - 2: standard error (write only)
        - As additional files are opened with open(2), creat(2), pipe(2), each file is assigned the lowest unused file descriptor
        - These integers are indices for a table of open files maintained by the OS for each process.
    - open() is a system call that attempts to open a file and returns an file descriptor
    - close() is a system call that attempts to close a file, make the file descriptor available

COMPUTER NETWORKING

Internet

- Hardware description:
    - The Internet is a network of **end systems** (a.k.a. hosts)
        - Ex: end systems include traditional PCs, servers, cell towers, laptops, phones
    - End systems are connected by a network of **communication links** and **packet switches**
- Services description:
    - The Internet is an infrastructure that provides **services** to **applications**
        - Ex: Applications include email, web surfing, messaging
    - These type of applications are called **distributed applications**: they involve multiple **end systems** that exchange data with each other
    - Apps run on end systems. They don't run on packet switches.

Communication Link

- Ex: Communication links include copper wire, optic fibre, radio spectrum
- Different links transmit data at different rates. I.e. They have different **transmission rates**
    - Transmission rates are measured in bits / second

Packet Switch

- Receives **packets** via an incoming **communication link** and forwards packets on its outgoing communication link.
- The two most common types of **packet switches** are **routers** and **link-layer switches**.
    - Link switches are typically used in **access networks**

- Routers are typically used in the **network core**

Transmission of Data

- When one **end system** transfers data to another end system, it segments the data and adds header bytes to each segment. The result is it sending **packets** of data to the destination end system, where packets are reassembled.

Route

- The sequence of **communication links** and **packet switches** that a **packet** takes to go from the beginning **end system** to the ending **end system**

Internet Service Provider (ISP)

- A network of **packet switches** and **communication links**
- Endsystems use **ISP**s to access the **Internet**
- An ISP network runs independently of other ISP networks, runs the **IP Protocol**, and conforms to naming and address conventions.

Protocol

- Protocol defines the format and order of messages exchanged between multiple communicating entities as well as the actions taken upon the transmission / receipt of a message or event
- Controls the sending and receiving of information in the **Internet**
- The two most prominent Internet protocols are **Transmission Control Protocol** (TCP) and **Internet Protocol** (IP)
    - The Internet's principal protocols are called **TCP/IP**
- Ex: HTTP, TCP, IP
- All communicating entities must be using the same protocol

Internet Protocol (IP)

- IP specifies the format of packets that are sent and received by end systems and routers

Internet standards

- Internet Engineering Task Force (IETF) is responsible for defining networking and protocols.

Application Programming Interface (API)

- A set of rules that an application on the sending end system must follow so that its data can be sent to an application of the receiving end system.

Network Edge

- Computers and other devices connected to the internet are called **end systems** because they are on the **Network Edge**
- End systems are divided into **client** and **server**

Access Network

- The physical network that connects an end system to its first router, called the **edge router**

Digital Subscriber Line

- Telephone companies offer internet access through the telephone line by dividing the data of downstream, upstream, and telephone channels into different frequencies
- Requires a DSL **modem** with ethernet connection

Cable Internet Access

- Cable television companies offer internet access through the same infrastructure that provides customers cable TV access
- Requires a Cable **modem** with ethernet connection
- Unlike DSL, users share access with each other. This means that

Fiber Optics

- More expensive, faster than cable

Satellite

- Able to reach users who do not have access to DSL, Cable, or FO

Dial Up

- Uses same infrastructure as DSL. Very slow

Local Area Network (LAN)

- A network that connects an end system with an edge router
- **Ethernet** is the most common LAN: copper wiring connects end systems with **Ethernet switches** that provide high speed within the LAN

Wireless LAN (WiFi)

- Wireless end systems locally send and receive packets to a **wireless access point** (base station) which is in turn wired to the rest of the Internet
- Ex: Wireless PC sends/receives packets to a base station, which is connected by Ethernet to a router, which is connected to a cable modem, which provides **broadband** access to the Internet

Wide Area Networks (WAN)

- Wide Area Networks connect end systems to base stations that are several kilometers away. Common examples include 3G and 4th Generational LTE.

Network Core

- The collection of **network switches** and **communication links** that connect end systems

Packet Switching

- In a network application, end systems send **messages** to each other in chunks called **packets**. Packets travel through **packet switches**, of which the two most common are **routers** and **link-layer switches**

Store-and-Forward Transmission

- Most **packet switches** use store-and-forward transmission at the inputs to **communication links**, which means waiting for all the complete reception of a packet before the sending of the packet via the link.
- Given N links (thus N-1 routers), L bits per packet, R rate of bits transmitted per second: Sending one packet has a delay of N * L / R
    - Delay increases as the number of links / routers needed grow

Protocol Layering

- **Protocols** can be organized in **layers**
- The **service model** of a protocol describes the services that a protocol offers to the layer above it. Each layer's protocol is implemented by having the protocol perform certain actions within the layer and use services of the the layer below.
- A protocol layer can be implemented in software/hardware of a combination of both
- The collection of protocols of layers make up the **protocol stack**
- Ex: The **Internet Protocol Stack** is a **network architecture** containing the application, transport, network, link, and physical layers

APPLICATION LAYER

Application Layer

- Contains network **applications** and their **protocols**
- Ex: protocol **HTTP** provides for document request and transfer, protocol **SMTP** provides for email transfer, protocol **FTP** provides for transfer of files between two end systems, protocol **DNS** maps human readable names of end systems to their 32-bit network address
- A **message** is a **packet** of information in the **application layer**
- In the **Internet**, application software is confined to the end systems

Application Architecture

- Designed by the application developer and determines how an application is structure throughout various end systems
- An app dev likely choose either **client-server architecture** or **peer-to-peer architecture** (P2P)

Client-Server Architecture

- There is an always-on end system called the **server** which **services** requests from many other end systems called **clients**
- Ex: Web **servers** service requests from browser running on **clients**
- Clients do not directly communicate with each other
- The server has a fixed, well defined address called **IP Address**
- Since the server is always-on and the server address is fixed and well defined, a client can always contact the server via sending a packet
- Ex: Web, FTP, Telnet, email

Data Center

- A collection of many end systems that act as one **virtual server** to serve many clients

P2P Architecture

- Direct intermittent communication between end systems called **peers** (typically PCs and Laptops)
- No reliance on dedicated servers
- Advantage over client-server architecture due to **self-scalability** since each peer adds service capacity to the system (as well as generating workload)
- Disadvantages
    - **ISP Friendly**: Residential ISP have asymmetric stream rates (high downstream, low upstream) but P2P architecture shifts the upstream provided by servers to residential peers
    - **Security**: It's harder to secure a distributed system compared to one server
    - **Incentives:** Peers need incentives to provide service capacity
- Ex: Bittorrent (File sharing app)

Process Communication

- A **process** is a executing program in an end system. Processes may communicate with each other
- Multiple processes on the same end system may communicate using **interprocess communication**, using rules governed by the end system's **operating system**
- Processes on different end systems communicate using **messages**

Client and Server Processes

- A **network application** consists pairs of **processes** that send **messages** to each other across a **network**
- Ex: In a web application, a client browser process exchanges messages with a web server process
- Ex: With P2P file sharing, the **peer** downloading the file is **client**, and the peer uploading the file is **server**. This implies a process can be both client and server

Socket

- The **socket** is the **interface** between the **process** and the **network**
- A socket is the **interface** between the **application layer** and the **transport layer** and is also referred to as the **API** between the **application** and the **network**
    - This means the application developer has control on the application side of the socket, but little control over the transport side of the socket
    - The app dev can choose which transport protocol (**TCP** or **UDP**) and specify supported transport layer parameters
- The **operating system** handles the transport side of the socket

Addressing Processes

- In order to identify a process, we require the address of the **end system** and the address of the **process** on that host.
- **IP Address** identifies the **end system** of a network and **port number** is used to specify the running application on that end system.

Services of the Transport Layer provided to the Application Layer

- The possible services that a transport layer protocol can provide the application can be classified as: **reliable data transfer**, **throughput**, **timing**, and **security**
- The network **Internet** uses the transport layer **protocols TCP** and **UDP**

TCP Services (Transmission Control Protocol)

- A **transport layer protocol** that is a **connection oriented service:** a **handshake** between **client** and **server** is performed before application-level messages begin to flow
- **Full-duplex**: Both connecting processes may send **messages** with one connection
- Does not provide security. Applications implement their own **encryption**, such as with **SSL** at the application layer

UDP Services (User Datagram Protocol)

- A **transport layer protocol** that is **connectionless**: no handshake before packet transfer
- Unreliable data transfer, no guarantee on ordering of packets
- No congestion control: sending side may send packets at any rate. However, this may be limited by intervening **links** or **congestion**

Services not provided by Internet Transport Protocols

- **Security** is implemented by SSL in the application layer
- **Timing guarantees** and **throughput** are not guaranteed, but this is fixed by clever design

Application Layer Protocol

- Defines how an application sends a **message** to another application
- Defines the types of messages, such as **request** messages and **response** messages
- Defines the syntax, semantics (meaning of) of message types and their **fields**

- Defines rules for how a process sends and responds to other messages
- Ex: **HTTP**, **SMTP**, Skype's proprietary app protocol

HTTP (HyperText Transfer Protocol)

- **HTTP** is implemented by a **client** program such as a **browser** and a **server** program such as a **web server**
- **Application protocol HTTP** uses the **transport layer protocol TCP**
    - Thus HTTP inherits secure message delivery
- **HTTP** is a **stateless** protocol. HTTP servers do not store **state** information on clients.
    - This means that if a client makes multiple HTTP requests to the server, the server cannot know that it has already served the client
- Ex: User clicks on hypertext -> browser (client) makes a HTTP request to the web server (server) -> the web server sends a HTTP response that contains the objects of the web page requested.
- The socket on the client end is the interface between the client process and the TCP connection. The socket on the server end is the interface between the server process and the TCP connection.
- Both client and server send and receive via their socket interface
- The **Web** uses client-server architecture. Web servers are always on, have fixed IP addresses, and can service requests from multiple clients.
- HTTP connections can be **non-persistent** where a connection must be made for each web page request. Or it can be **persistent** where a single connection is made for multiple web page requests

Web Page (a.k.a. document)

- Consists of **objects**. Objects are **files** such as HTML/JPEG/CSS files
- Most web pages consist of a base HTML file and several referenced objects
- Objects are referenced using **URLs**
- Ex URL: http://hostname/path/to/file
- Ex Client: Google Chrome; Ex Server: Apache

HTTP Message Format

- Two types of HTML message formats: **request** and **response**
-

Transport Layer

- In the **Internet**, the two transport **protocols** are **TCP** and **UDP**, both of which can transport **application-layer messages**.

- Ex: protocol TCP is connection oriented as it guarantees **reliability** in the delivery of messages and **flow control** (speed matching). TCP breaks long messages down into shorter **segments** and provides **congestion-control** by throttling source transmission when the network is congested
- Ex: protocol UDP is connectionless service to applications and provides no reliability, no flow control, no congestion control.
- A **segment** is a **packet** in the **transport layer**

Network Layer (a.k.a. IP Layer)

- Responsible for moving **datagrams** from one end system to another
- Given a **segment** from a source **transport layer**, the **network layer** delivers the segment to the destination transport layer
- Ex: protocol **IP** defines **fields** in the datagram and how routers and end systems act on these fields. All Internet components that have a network layer must run the IP protocol
- Ex: numerous **routing protocols** determine the **route** that datagrams take within the network
- A **datagram** is a **packet** in the **network layer**

Link Layer

- Moves one packet from one **node** (host or router) to another node
- Given a **datagram** from the source **network layer**, the **link layer** delivers the datagram to the destination network layer
- Some link layer protocols provide reliability (in moving one packet from one node to another node). This differs from TCP which provides reliability only between end systems
- Ex: Link layer protocols include **Ethernet**, **WiFi**, cable access network's **DOCSIS**
- Datagrams may traverse several links, meaning that a datagram may be handled by multiple link layer protocols
- A **frame** is a **packet** in the **link layer**

Physical Layer

- Moves individual **bits** within the **frame** from one **node** to the next
- Again, the protocols used in the **physical layer** depend on the **link layer** and the **transmission medium**
- Ex: link layer protocol **Ethernet** has multiple physical layer protocols it may use as a **service** such as twisted-pair copper wire, coaxial cable, optic fiber, etc. In each case, the bit is moved in a different way

OSI Model

- 7 Layers: Application, Presentation, Session, Transport, Network, Link, Physical
- The role of the **presentation layer** is to provide services to applications to interpret the meaning of exchanged data
    - Ex: Data description, encryption, compression
- The role of the **session layer** is to provide synchronization, checkpointing, and a recovery scheme
- The **Internet protocol stack** misses the presentation layer and session layer, and depends on the developer of the **application** to implement features present in these layers

Encapsulation

- At each layer, a **packet** has two types of fields: **header field** and **payload field**
- The header field typically contains info on how deliver the packet from the current layer to the layer above and info relevant to implementing the current layer.
- When a packet is traversing down the **protocol stack**, each layer appends header information on the current packet with the resulting packet being an encapsulation of packet information of the current layer and the layers above it
- The header information is used to direct the packet traverse up the protocol stack
- Ex: **packet** from **application layer** is called **message**. The transport layer **encapsulates** the message by appending **transport layer header** information that results in a transport layer **segment**

World Wide Web

- An **information space** where documents where **documents** and other **web resources** are identified by **URLs**
-

Stateless Protocol

- A **communications protocol** in which no information is retained by either sender or receiver. This means that they are agnostic of the state of one another.
    - Ex: A **UDP** connectionless session is a stateless connection because the system doesn't maintain information about the session during its life.

URI

- **Uniform Resource Identifier** is a string that provides a simple and extensible way to identify a **resource**.

    - It is **uniform** in that it's syntax has uniform interpretation with respect to different types of URI's.
    - It is related to a **resource** in that a resource is a mapping to an entity (a.k.a. content).
        - A resource can remain constant with respect to changes to entities.
    - A **identifier** is an object that can act as a reference to something that has an identity. In the context of URI, the object is a sequence of characters with a restricted syntax.
    - Ex: www.site.com/text.txt is a URI because it is an identifier that references this resource. The resource maps the string "www.site.com/text.txt" to the entity text.txt located at server site.com
- URL and URN are subsets of URI

URL

- **Uniform Resource Locator** is URI that identifies resources via a **primary access mechanism** (ex: network location), rather than by name or by attributes of that resource

URN

- **Uniform Resource Name** is URI that is **globally unique** and persists even when the resource ceases to exist or is unavailable.
- Ex: urn:isbn:12345 is the international standard book number that uniquely identifies a book with the isbn 1234

OPERATING SYSTEMS

Key Ideas

- **Virtualization**: creating and manipulating abstractions from physical entities
- **Concurrency**: effect of the system does not depend on order of execution
- **Persistence**: state of a system outlives the process that creates i

Module

- A **module** is a component of a system whose internal functionality doesn't need to be understood in order to utilize the module's **services**. Modules **encapsulate implementation** details, and only a knowledge of the **interface** is required
- A system is **modular** when all components are **modules**

Good Modularity

- Any component is small or can be broken down. The purpose of any component is clear.
- Functions that work on the same resources and closely related functionalities should be combined into a single **module**. This principle is called **cohesion**

- Breaking down a system into components increases overhead cost for communication between components. Know when to combine functionality within one module

Running a Program

- To run a **program**, the **processor fetches** an **instruction** from **memory**, **decodes** the instruction, **executes**

Operating System

- An **operating system** (OS) is a collection of software that makes running **programs** easy, allow programs to share **memory**, allow programs to interact with **hardware devices**, etc
- People who make **applications** are the **users** of the OS. They use the OS's **API**, called **system calls**, which allow user programs to indirectly access **devices**. System calls provide more control and security than to directly give applications control of a system's devices
    - The OS's system call API is a **standard library** to applications
- The OS is a **resource manager** as it **manages** resources such as CPU, memory, disk

Virtualization

- The **abstraction** that the OS provides makes it a **virtual machine**, as it virtualizes physical resources into their virtual form
    - The OS virtualizes the **CPU**: you can run many processes through many virtual CPUs even when you have limited physical CPUs
    - The OS virtualizes **memory**: each process has its own **virtual address space** (or just address space) entirely to itself
- The OS does not virtualize the **disk**, since it useful for multiple processes to share it

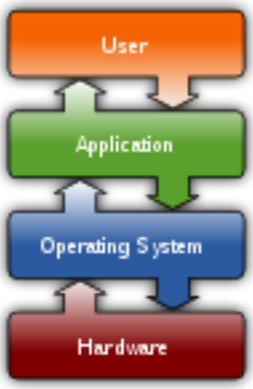
Concurrency

- Since the OS needs to handle many tasks at once, it runs into the problem of **concurrency**

Persistence

- The **file system** is part of OS software that manages **persistent data**.
    - This means that the file system manages the **disk**
    - File systems use intricate write protocols such as **journaling** and **copy-on-write** to deal with system failures

Operating System Services



- The **OS** services the **application layer**
- **CPU / Memory Abstractions**: processes, threads, virtual machines, virtual address spaces, shared segments
- **Persistent Storage Abstractions**: filesystems, files
- **Other I/O Abstractions**: virtual terminal sessions, windows, sockets, pipes, VPNs, signals (as interrupts)

Instruction Set Architecture (ISA)

- The set of instructions a computer supports
- ISAs differ in word length (8 vs 16 bit), features (low power, floating point), design philosophies (**RISC** vs **CISC**)
- ISAs come in families (Pentium, x86) and are often backwards compatible
- The ISA divides the instruction set between **privileged** and **general instructions**

Privileged vs General Instructions

- Any code running on a machine can execute **general instructions**
- **Processor** must be put into special mode to execute **privileged instructions**

System Calls vs Procedure Calls

- **System calls** raise the **hardware privilege level** and transfer control to the OS
- **Procedure calls** do not require this privilege and thus don't access hardware
- User applications run in **user mode**, which means hardware restricts what the application can do.
- System calls look like procedure calls because they are. However, they are implemented in the hardware using the **trap instruction**

Trap

- Typically a hardware **instruction** that triggers a **system call**. The hardware transfers control to a **trap handler** which raises the **privilege level** to **kernel mode**. After the OS services the request, it executes a **return-from-trap** instruction which lowers privilege level and returns control back to user process

Platform

- The **ISA** doesn't completely define a **computer**
- The functionality beyond user mode instructions such as with interrupt controllers, MMU, I/O bus, I/O devices are independent of ISA
- A **platform** is the combination of ISA and other functions that make up a computer

Binary Distribution Model

- The OS is distributed via **binary** code, not **source** code. **Device drivers** are added afterwards to support different platforms.

Device driver

- Software that controls a particular hardware device of the computer (ex: keyboard)

Device file (a.k.a. special file) (Unix-like)

- An **interface** to a **device driver** that appears as an ordinary file on the filesystem
- These files let applications interact with hardware devices by using its device driver via **system calls**

Terminal Mode

- One of a set of possible states of a terminal **character device**
- Determines how characters are interpreted

Daemon

- A process that runs in the background, rather than being directly controlled the user.

Graceful Transition

- A **switch** (ex: context switch) that hides the fact that a resource used to belong to someone else

Library

- **Libraries** are **non-volatile resources**.
- They are convenient in that only a single well maintained copy is required
- They have multiple **bind** time options (static, shared, dynamic)

Static Library

- Bind a l

PROCESSES

Process

- Informally, a **process** is a running program
- A process is an **abstraction** (virtualization) that allows the system to run many different applications with limited number of processors

Context Switch

- Stopping a process and starting another on a CPU

- A **context switch** is a **mechanism** in that its effect does not depend on the choice of when to switch or what processes to switch

Scheduling

- Given a number of processes ready to run, which should be run. This choice differs depending on the goal (ex: interactivity vs most work done)
- Thus the optimal choice of **scheduling** is a matter of **policy**

Machine State

- What a **process** can read or write to
- A process's **machine state** includes its address space, registers, and persistent memory

Address Space

- The **memory** that a **process** can **address** (has access to)
- A process's **code** and **data** are stored in **memory locations**. The process's set of memory locations is called its **address space**

Registers

- The **program counter** (a.k.a. **instruction pointer**) register stores the address of the executing instruction
- **Stack pointer** and **frame pointer** are used to manage the **run-time stack** (a.k.a stack). The stack contains the local variables, function parameters, and return addresses

Process API

- Modern OSes allow **applications** the creation, deletion, waiting (stopping execution), miscellaneous control, and getting status of processes

Process Creation

- A **process** is created by writing the associated code and data from **disk** to **memory**. This is because the processor can only execute the process's instructions if they are in memory
- Modern OSes **lazily** load by only writing code and data if they will be required during program execution. Lazy loading is implemented by **paging** and **swapping**
- The OS will also allocate memory for the program's **heap**. i.e. Instead of knowing the values of what will be written to memory, the OS reserves memory as part of the program's heap.
- (Unix) Each process by default gets three open **file descriptors** (for I/O), for standard input, output, and error
- After loading data and code into memory, initializing the stack and heap, and setting up I/O, the program starts running at the **entry point**: main()
    - Doing this means the OS **transfers control** to the created process

Process States

- A **process** is in one of three **states**:
    - **Running**: A process is running on the processor and executing instructions
    - **Ready**: A process is ready to run, but the OS has not chosen to run it
    - **Blocked**: A process has performed an operation that makes it unready to run until an event takes place.
        - Ex: A process initiates an I/O request to disk and is blocked so that another process may run and use the processor
- Some other states:
    - **Initial**: A process that has just been created
    - **Final** (zombie in UNIX): A process that has exited but has not been cleaned up

Process State Transition

- The part of the **OS** that manages when processes are being run is the **scheduler**
- A process moving from **ready** to **running** is **scheduled**
- A process moving from running to ready is **descheduled**

Process List

- A **data structure** the OS maintains to keep information on **processes**
    - Ex: The **register context** holds register values for a stopped process, useful for **context switches**

Shell (Unix)

- A **user program** that shows you a **prompt** and waits for input
- When you run a program via the **shell**: the shell calls fork() to fork a child process, calls a variant of exec() to run the specified program, and calls wait() and waits for the child process to complete. When the child completes, the shell process returns from wait(), prints a prompt again and waits for input
- The separation of fork() and exec() gives a lot of control to the shell
    - **File redirection** to a new file is accomplished by closing the child process's standard out **file descriptor**, having the child open a new file and use closed file descriptor

Pipe (Unix)

- The output of one process is output to a **pipe** and the input of another process is connected to the same pipe
- Implemented with a **queue** data structure

INTER-PROCESS COMMUNICATION

Inter-process Communication

- A **process** can be either an **independent process** or a **cooperating process**
    - An independent process is not affected by the execution of other processes
    - Many processes cooperate to increase computational speed, convenience and **modularity**
- **Inter-process communication** (IPC) is a mechanism that allows processes to communicate and synchronize their actions, and is seen as a method of **cooperation** between them.
- Processes can communicate either by **shared memory** or by **message passing**. An **operating system** can implement both kinds of inter-process communication.

Inter-process Communication: Shared Memory

- Ex: Bounded Producer-Consumer problem: One process is producer, the other is consumer. The two processes **share memory** in a place called a **buffer**. The buffer is bounded, meaning that the producer process can produce up to a certain amount of data before needing it consumed.

Inter-process Communication: Message Passing

1. Processes establish a **communication link**. If a link already exists, no need to reestablish.
2. Processes exchange messages using basic primitives, send and receive
- A message can have header and body

MECHANISM: LIMITED DIRECT EXECUTION

Problem #1: Restricted Operations

- How do we let user programs use devices properly?

Time Sharing

- In order for the **OS** to **virtualize** the **CPU**, it lets processes take turns running.
- We wish to have **high performance** by minimizing the overhead created from **time sharing**
- We wish to have the OS retain **control** over the CPU so that user processes can't ruin the system

Direct Execution

1. The **OS** runs the program by creating an entry in the **process list**, allocates **memory**, loads program into **memory**, set up stack with argc/argv, clears registers, and executes **call** main()
2. The **program** runs main(), then executes **return**
3. The **OS** frees the memory of the process and removes the process from process list
- **Direct execution** is fast, but it doesn't allow us to perform **restricted operations**

    - Ex: If we let a program perform I/O, then all notions of **privacy** is lost

Processor Mode

- The **processor** is either executing instructions in **user mode** or **kernel mode**. User programs run in user mode and the OS runs in kernel mode
- Code running in **user mode** is limited to what it can do.
    - Ex: User mode code can't issue an I/O request. Typically, this creates an **exception** which leads to the OS killing the process
- OS code running in **kernel mode** may execute general and privileged instructions, such as issuing I/O requests
- User programs use fun

Kernel Stack

- Stores caller saved **registers** for the duration of the **system call**
- **Trap** instruction causes this storing. **Return-from-trap** instruction causes the popping of these values

Trap Table

- Contains the location and associated **system-call numbers** of **trap handlers**
    - This usage of making programs use system-call numbers instead of letting them directly access trap handlers is a way of limiting the control of programs and having the OS supervise their intentions
- When the OS handles a system call, it looks up the **trap handler** via the **trap table**
- The hardware instructions that locate the trap table are **privileged**.

Limited Direct Execution

1. The OS at boot time initializes the **trap table**
2. After boot time, the OS creates a user process by adding an entry to the **process list**, loading the program into **memory**, set up the user and **kernel stack**. Up until this point, the processor is still in **kernel mode**. Thus to begin running the user process, the CPU executes a **return-from-trap** instruction
3. This instruction causes the hardware to restore the contents of the registers in the **kernel stack**, switch to **user mode**, and jump to the program's main()
4. The program runs main(). When it wishes to perform a restricted operation, it makes a **system call**, which causes a **trap** to occur
5. The hardware saves register values to the **kernel stack**, switches to **kernel mode**, and jumps to the **trap handler** via the **trap table**
6. The **trap handler** (part of OS) performs tasks associated with the system call, and eventually the processor executes a **return-from-trap** instruction
7. The hardware pops the register values from **kernel stack** to the thread's **registers**, switches to **user mode**, and jumps to the value of the **program counter** register (typically the next instruction)
8. Eventually the program **returns** from main

9. The OS frees the memory of the terminated program, and removes the process from the **process list**

Problem #2: Switching Processes

- The OS is not running while a user process is running. The OS manages the switching of processes. Thus how can we switch processes when a user process is running?

Cooperative Approach

- The OS expects processes to be **cooperative** and **yield** (via system call) periodically so that the OS can transfer control to other processes
- This approach is vulnerable to malicious design and accidents (bugs)

Non-Cooperative Approach

- A timer device is programmed to raise a **timer interrupt** that an **interrupt handler** (initialized during OS boot time) responds to. The timer interrupt causes a switch to **kernel mode** and thus the OS can switch processes
- The instruction to start/stop the timer is privileged and occurs at boot time
- **Interrupts** are similar to **system calls** in they also require **register** states to be saved onto the **kernel stack**

Context Switch

- If the **scheduler** decides to switch the running process on the CPU, then the system performs a **context switch**
- This causes the old process's **register** values to be saved on its **kernel stack**. Then the next running process's restores its registers by popping register values off from its kernel stack
- There are two types of register saves/restores that happen here:
    1. Whenever an **interrupt** occurs, the hardware implicitly saves the registers of the running process to its **kernel stack**
    2. Whenever the OS decides to perform a **context switch**, the software explicitly **saves** the registers of the currently running process to its **process structure** and restores the registers from the **kernel stack** of the process to be run next

SCHEDULING

Workload Assumptions

- **Workload** is the set of processes running in the system
- **Jobs** are processes

Scheduling Metrics (types of measurement)

- **Turnaround time** is the time at which a **job** completes minus the time the job arrived. Turnaround time is a **performance metric**
- **Response time** is the time at first run - time when job arrived

Performance vs Fairness

- Often there is a tradeoff between these two qualities that we desire

Non-Preemptive vs Preemptive Scheduling

- **Non-preemptive scheduling** means the system runs each job to completion before considering the next job
- Virtually all modern OSes are **preemptive**, meaning that the OS stops processes and performs **context switches**

First In, First Out (FIFO) (non-preemptive)
- In general, average **turnaround time** is often poor: what if a long job is run first?
- To minimize this, we can run the shortest job first

Shortest Job First (SJF) (non-preemptive)
- Run the shortest job, then so on…
- But: Suppose you have 1 long job. While you're running it, a short job comes. This is nonoptimal.

Shortest Time-to-Completion First (STCF) (preemptive)
- Whenever a new job appears on the system, determine the job that has the least time left (to account for the new job)
- Good for **turnaround time**, bad for **response time**

Round Robin (RR) (preemptive)
- Instead of running jobs to completion, RR runs jobs for a **time slice** (called **scheduling quantum**), then switches to the next job in the **queue**
    - The length of the **time slice** must be a multiple of the **timer-interrupt** period
- If the **time slice** is too small, then the cost of the **context switches** become too great.
- Thus there is a **policy** decision in balancing between **response time** and **turnaround time**
- RR essentially stretches out each job as long as it can, increasing turnaround time
- Good for **response time**, bad for **turnaround time**

Incorporating I/O
- Whenever a process requests I/O, it typically becomes **blocked**. It is not optimal to wait for the I/O request
- Furthermore, we wish to **overlap** operations as much as possible (not be idle)

No Oracle Problem
- Our **scheduler** doesn't know about the length of each **job**

SCHEDULING: MLFQ
- This topic is about building a progressively better MLFQ

Multi-Level Feedback Queue (MLFQ)

- Attempts to achieve an optimal balance between **turnaround time** (performance) and **response time** (interactivity)
- Has multiple distinct **queues**, each with different **priority levels**. A job ready to run is on a single queue. MLFQ uses **priority** to decide which job to run
- **Rule 1**: If Priority(A) > Priority(B) Then A runs, B doesn't
- **Rule 2**: If Priority(A) = Priority(B) Then A, B run in RR
- MLFQ varies **priority** based on **observed behavior**
- Ex: If a program repeatedly yields its CPU while waiting for I/O, then MLFQ will keep its priority high, because this is how an interactive program might behave
- Ex: If a program uses the CPU intensively for long periods of time, then MLFQ will keep its priority low

MLFQ Changes Priority

- Recall that our **workload** consists of **interactive** jobs that are short running, and **CPU-bound** jobs that are long-running but don't require good response time
- **Rule 3**: When a job enters the system, it is placed at the highest priority
- **Rule 4a**: If a job uses up an entire **time slice** while running, its priority is reduced
- **Rule 4b**: If a job gives up the CPU before the **time slice** is up, then its priority stays the same
- **MLFQ approximates SJF** because longer jobs naturally descend to low priority and are run after short jobs
- **MLFQ handles I/O** by Rule 4b. Jobs that request for I/O give up their CPU before their time slice occurs, and thus they remain at the same priority

MLFQ Boosts Priority

- We need to avoid **starvation:** too many **interactive** processes consuming all the CPU time (leaving no time for others)
- We need to account for **changes in behavior**: a process may change between being CPU intensive and interactive
- **Rule 5**: After some time period S, move all jobs of the system to the topmost queue
    - If S is too high, then CPU-bound jobs starve more. If S is too low, then interactive jobs aren't getting enough CPU time
- MLFQ **boosts** the priority of all jobs periodically. This means CPU intensive processes don't **starve** even when there is too many interactive processes and a **CPU-bound** job that has become **interactive** will be recognized (and not ignored at the bottom of the queue)

MLFQ Accounts for CPU Time

- We need to avoid processes **gaming the scheduler**: processes monopolizing the CPU such as by making I/O requests at 99% of the time slice
- Replace Rule 4a, 4b with:
- **Rule 4**: Once a job has used up its time allotment at a given level (regardless of how many times it has yielded the CPU), its priority is reduced (moves down the queue)

VIRTUALIZING MEMORY

Goals

- **Transparency**: The OS should virtualize memory in a way that can't be perceived by user processes. This makes programming user programs easier
- **Efficiency**: In terms of both **time** by not making programs run too slowly and **space** by not using too much memory for structures needed for the overhead of virtualization.
- **Protection**: The OS should protect processes from each other. Further, it should protect processes from the OS itself
-

Address Space

- The abstraction of **physical memory**
- Consists of **code**, **stack**, **heap**
- Code is static, as its size isn't variable. Stack and heap may vary in size during runtime.

Principle of Isolation

- When two processes are **isolated** from each other, this implies that the failure of one does not cause the failure of the other.
- The OS uses **memory isolation** to prevent processes from interfering with each other

Every Address You See is Virtual

- In the eyes of the programmer for the user program, addressing is always virtual. It never deals with physical memory

MEMORY API (UNIX)

Types of Memory

- **Stack** (a.k.a. **automatic memory)**: Allocations and deallocations are managed implicitly by the **compiler**
    - Ex: void func() { int x; … } //x is allocated on the stack, deallocated when the call to func returns
- **Heap**: long living memory where allocations and deallocations are managed explicitly by the **programmer**

sizeof()

- A **compile-time operator**: the argument to operator sizeof is known at **compile time**
    - Contrast this with a **function call**, which accepts arguments whose value is known at **run time**

- If you pass in a variable, it returns the size of the variable's type. Thus passing in a pointer to heap will result in it returning the size of the pointer and not the number of bytes allocated in heap memory

malloc()

- Pass in a size you want of heap memory, then it either succeeds or fails in returning a pointer to the newly allocated space

free()

- Free heap memory no longer in use
- To implement free(), the memory-allocation library has to keep track of allocations

Error: Forgetting to Allocate Memory

- Many **routines** expect memory to be allocated before they are called
- Ex:
  char *src = "hello";
  char *dst; // oops! unallocated
  strcpy(dst, src); // segfault and die
- The **segmentation fault** above occured because we didn't allocate enough memory for dst. src pointed towards an array of 6 chars ("hello\0"). dest wasn't allocated memory to use as a **buffer** for the call to strcpy

Error: Not Allocating Enough Memory

- A similar error is not allocating enough memory, called a **buffer overflow**
- Ex:
  char *src = "hello";
  char *dst = (char *) malloc(strlen(src)); // too small! (forgets the /0 character)
  strcpy(dst, src); // work properly
- Often it runs properly, because there is often extra padding provided
- But this is still dangerous and there are many security vulnerabilities associated with this

Error: Forgetting to Initialize Allocated Memory

- When you allocate memory, you need to remember to write it before you read. Else you get an **uninitialized read**

Error: Forgetting to Free Memory

- Known as a **memory leak**. Slow leaking memory in a program leads to it running out of memory, and then a restart is required
- In general, when you are done with a chunk of memory, you should free it.
- A **garbage collector** cannot free memory if **references** to it still exist. This means memory leaks can still occur!

Error: Freeing Memory Before You are Done With it

- This mistake is called a **dangling pointer**. When you free memory and then try to access the memory again, you may crash or overwrite valid memory that you shouldn't

Error: Freeing Memory Repeatedly

- Known as **double free**. Undefined behavior

Error: **Calling Free Incorrectly**

- Known as **invalid free**. free() only expects pointers that have been returned by malloc()

Levels of Memory Management

- **Memory management** is performed at the program level and the OS level
- At the program level, calls to free() will deallocate memory
- When a program exits, memory management is also performed at the OS level. The OS deallocates all memory that was allocated for the program. Thus even if the program forgets to free memory itself, the OS will resolve the memory leak
- Implications: It's okay for short lived programs to have memory leaks, and bad for long living programs such as **web servers**

Memory Management Tools

- Purify, Valgrind

Underlying OS Support

- malloc(), free() are **library calls** that manage space within the **virtual address space**. They rely on **system calls** that ask the OS for memory allocation/deallocation
    - Ex: brk(), sbrk() are system calls used to change the program's **break**, the end of **heap** location
    - Ex: mmap() system call creates an **anonymous** memory region within the program that is associated with **swap space**

calloc()
- Allocate memory and zeroes the values

realloc()
- Pass in pointer to memory you've already allocated. It will allocate a new space, copy values from old to new, and free the passed in pointer.

MECHANISM: ADDRESS TRANSLATION

Address Translation (a.k.a. **hardware-based address translation**)
- The mapping from a **virtual address** to a **physical address**
- The **hardware** transforms each **memory access** (ex: instruction fetch, load, store) by mapping the **virtual address** provided by the instruction to a **physical address**. In general, an **address translation** is performed on every **memory reference**
- Hardware alone isn't enough to perform address translation. The OS is involved in **memory management** by keeping track of free and in-use memory locations

Dynamic (Hardware-based) Relocation (a.k.a. **base and bounds**)
- Two registers per CPU: base register and bound register that let us place the user **address space** anywhere in **physical memory**. This also ensures that the process can only access its own address space
    - The **base** register transforms virtual addresses to physical addresses
    - The **bound** register helps the processor check that a process's memory reference is within the bounds of its address space. Processes that try to access outside of their address space raise an exception called a **fault**
    - The base and bound registers are kept on chip, **one pair per CPU**
    - The part of the of the processor that helps with address translation is called the **Memory Management Unit** (MMU)
- Resulting **translation**: physical address = virtual address + base
- Each memory reference is a **virtual address**. After translation, it becomes a **physical address**
- This relocation of the address space occurs at **runtime** (therefore dynamic) and we can move the address space even after its running
- **Address translation can be handled by hardware with no OS intervention**
    - This means it's very fast!

Free List

- Data structure created by the OS to track which parts of **physical memory** is free

Memory Management Unit (MMU)

- Contains the **base** and **bound** registers. It is the hardware responsible for **address translation**

Hardware Support

- Modifying the **base** and **bound** registers requires **privileged instructions**
- CPU generates **exceptions**, when memory is accessed illegally by being **out of bounds**
- The OS has **exception handlers** that run when these exceptions are raised

Hardware Requirements

- **Privileged mode** to prevent user-mode processes from running privileged instructions
- **Base/bound registers** to support **address translation** and **bounds checks**
- Privileged instructions that update base/bounds, register exception handlers, and raise exceptions

OS Requirements

- **Memory management** in the allocation, deallocation of memory via the **free list**
- **Base/bound management** in the proper storing and restore during **context switches**
- **Exception handling** by providing **exception handlers** that run when an exception occurs.
    - The handlers are installed at **boot time** via privileged instructions

Internal Fragmentation

- Space that is not used inside of **allocated memory** results in **internal fragmentation**
- Ex: Internal fragmentation between heap and stack when they are not large enough

Problems in this Basic Address Translation

- The user's **address space** may not be placed contiguously in **physical memory**
- The size of the user's address space may exceed the size of physical memory

FREE SPACE MANAGEMENT

- How do we manage free space and avoid fragmentation?
- Specifically, we focus on **external fragmentation**

External Fragmentation

- Free space between allocated spaces.

- **External fragmentation** is problematic because we may be unable to allocate a contiguous chunk of memory when every free chunk is individually too small, even though the sum of free chunks is enough

SEGMENTATION

Segmentation

- Instead of having a base and bounds pair for the process, have a base and base pair for each logical **segment** of the process: Code, Stack, Heap
    - With **segmentation**, we eliminate the potentially large amounts of unused space between the stack and the heap
- The etymological origins of the **segmentation fault** comes from the illegal memory access of segmented machines

Which Segment are We Referring to?

- How does the hardware know which segment we refer to when it is translating a memory address?
- **Explicit** approach: Of the 14 bits used for the address, use the top 2 bits to distinguish between code, stack, and heap. Use the lower 12 bits to as the offset for the associated segment
    - Ex: If top 2 bits are 00, it could signify the code segment. Thus to translate this address, add the 12 bits (the offset) to the base register of the code segment. Check for in bounds by confirming that its less than the bounds register
- **Implicit** approach: The hardware determines segment based upon whether the associated instruction was a **fetch** (Code), used the **stack or base register** (Stack), or any other address (Heap)

What about the Stack?

- Since the stack grows backwards, we inform the hardware of the direction a **segment** grows by using a bit: 1 for growing upwards, 0 for growing downwards

Code Sharing

- The hardware supports sharing via **protection bits**, which are bits at the start of the segment that describe whether or not the segment can be read, written, or executed
- This allows us to map one physical segment (such as code) to many virtual address spaces and save memory
    - This is feasible when segments aren't writable

Fine-grained vs Coarse-grained Segmentation

- **Coarse-grained segmentation** is the segmenting of processes via their logical segments: Code, Stack, Heap
- **Fine-grained segmentation** consists of segmenting a large number of small segments and creating a map via **segmentation table**

Compacting Memory

- **External fragmentation** will always occur despite our techniques.
- One way to get rid of it is via **compacting memory**, where fragmented memory is moved into a contiguous area in memory. Then update the base and bound registers to reflect the changes in the segment locations.
- Compacting memory removes external fragmentation, but is very costly for the CPU

PAGING

Problems with Segmentation

- **Segmentation** involves the division of memory into **variable-sized** pieces. This leads to **external fragmentation**

Paging

- A **fixed-sized** approach to **space-management** is **paging**, which eliminates **external fragmentation**
- We divide a process's **virtual address space** into fixed size pieces called **pages**. We view **physical memory** as an array of fixed sized slots called **page frames**. Each page frame corresponds to a page
- Advantages of paging is **flexibility**: the availability of memory is not dependent on the choice of which pages to allocate / deallocate
- The OS keeps a **per-process page table** data structure: it stores **address translations** that map virtual pages of the address space to pages in physical memory
- A **virtual address** consists of a **virtual page number** (VPN) and an **offset**
    - The location of this virtual address in the virtual address space is itself
    - The location of this virtual address in both the virtual address space and physical memory is described by VPN and offset. VPN describes the page that the virtual address is in. Offset describes how far the location of the virtual address is from the start of the page it is in.
- The virtual address is mapped to a **physical address**, which is described by a **physical page number** (PPN) and an **offset**. Essentially, addressing via **pages** is a map from VPN to PPN. The offset stays constant

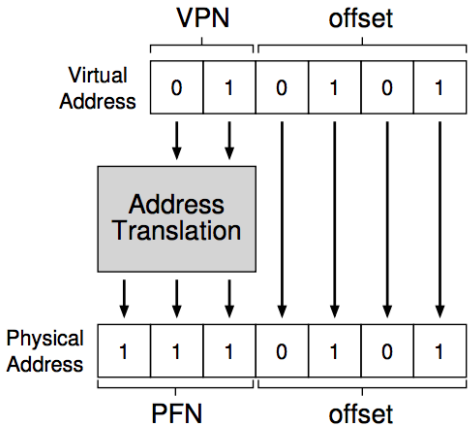


Figure 18.3: **The Address Translation Process**

Page Tables

- A data structure that maps virtual addresses to physical addresses. Since offsets remain constant, it is really a map of virtual page numbers to physical page numbers.
- Consists of **page table entries** (PTE)
- **Linear page table** is the a **page table** with array data structure. The OS indexes the linear page table by virtual page numbers and looks up (maps) the corresponding physical page number

Page Table Entry (PTE)

- **Page table entries** are elements of **page tables**
- Contains a **valid bit**, a bit that indicates whether or not the translation (of the page) is valid. If the page contains Code, Stack, Heap of that particular **process** then the translation is valid. If the page is unused (unallocated) for that particular **process**, then the translation is invalid.
    - This is used to save a lot of memory, especially in **sparse** address spaces

- Contains a **protection bit**, which describes whether or not the page can be written by that particular process
- Contains a **present bit**, which describes whether or not the page is present in memory, or if instead it is on disk
- Contains a **dirty bit**, which indicates if page has been modified since it was brought into memory
- Contains a **reference bit**, which indicates if the page has been accessed. This is critical during the selection process of **page replacement**
- **Exceptions** are raised when processes try to access certain pages that conflict with the status of these bits

Paging: Too much memory

- Unlike with **segmentation**, dividing memory into pages can be very costly

Paging: Too Slow

- Referencing memory via **paging** requires an extra memory reference on the **page table** itself (to find the mapping from virtual address to physical address)
- We need to design the **hardware** and **software** to

TRANSLATION-LOOKASIDE BUFFER

Motivation

- Using paging to support virtual memory can create large overhead in both space (page tables are large) and performance (page table lookup is an extra memory reference)
- To help the OS in this task, we use **hardware** called **TLB**. We use it in the hopes that most memory references will be made in cache and not in the **page table** in main memory

Translation-Lookaside Buffer (TLB)

- The **TLB** is part of the **MMU**, and is a hardware **cache** of popular virtual-to-physical address translations. A better name would be **address-translation cache**
- During each memory reference, the hardware first checks the **TLB** before checking the **page table**.
- Thus the TLB is great for reducing the performance overhead of **paging**

TLB Basic Algorithm

1. A memory reference is requested
2. Extract the VPN from the virtual address and check if the the TLB has translation
3. If it does, then **TLB hit**: we known PFN, can concatenate with offset, and access memory (end result: 1 reference to main memory)

4. If it doesn't, then **TLB miss**: we do extra work in making a memory reference to the **page table**.
5. After finding the PTE in the page table, we store the PTE into the TLB.
6. We repeat the algorithm with the same memory reference (this time, we're ensured a **TLB hit**)

- The TLB, like all **caches**, is built on the premise that in the common case, translations are found in the cache (mostly hits) (end result: 2 references to main memory)

Spatial Locality

- If a program accesses memory at x, it will probably access memory near to x
- Ex: Memory references during the iteration over elements of an array

Temporal Locality

- Making a memory reference to x makes it more likely we'll access x again
- Ex: After the iteration of the elements of the array, the program will probably perform another memory reference concerning the array
- Ex: Loops, loop variables

Who Handles TLB Miss?

- The **hardware** may raise an **exception** (trap) and let the OS deal with the **TLB miss** via a **trap handler**
- The TLB miss return-from-trap instruction is different from a system call return-from-trap. Instead of the hardware setting the PC to be the next instruction, the PC must be set to the current instruction (the memory reference that failed)
- When handling a **TLB miss**, the OS needs to be careful not to cause an infinite chain of TLB misses. This can be done by reserving some physical memory for TLB miss handlers only. The OS **wires** a register to reserve some TLB entries for the OS

TLB Contents

- A typical **TLB** has 32, 64, 128 entries, and is **fully associative**
    - This means that we can access all entries of the TLB in **parallel**
- A TLB entry looks like: VPN | PFN | other bits

Context Switching

- Problem: When we perform a **context switch** from A to B, we need to avoid the situation where B thinks that a entry in the TLB is B's when it is A's. This can happen if the TLB entry **valid bit**, which indicates if the virtual page is in the TLB, is not updated during the context switch
- One solution is to **flush** the TLB during a context switch, which sets all valid bits to 0
    - If the system context switches a lot, then the associated overhead is high
- Another solution is to add **hardware** support via an **address space identifier** (ASI), which uniquely differentiates the pages of different processes

Replacement Policy

- During **cache replacement**, the goal is for the OS to **replace** an entry such that the system minimizes **miss rate**
- **Least-recently used** (LRU): take advantage of **locality**
- **Random**: simple, and avoids corner cases
    - Ex: Suppose the program loops over n+1 pages and the TLB contains n pages. Then policy LRU misses every page and policy random does better

TLB Instructions

- For systems in which the TLB is software managed, there must be privileged instructions to manage the TLB

Culler's Law

- **Random access memory** implies that any part of RAM can be accessed with the same speed as another part. This is not true. The speed of accessing the page depends on whether or not the **address translation** is stored in the TLB cache

BEYOND PHYSICAL MEMORY: MECHANISM

How to go beyond physical memory

- Typically, the size of **physical memory** (and thus # of physical pages) is far less than the size of **virtual memory** (# of virtual pages). We use a **secondary storage device** such as a hard disk drive for extra pages that can't fit in main memory

Swap Space

- The space that we reserve on **disk** (secondary storage) is called **swap space**, because we swap **pages** from memory to it and we swap pages from it to memory
    - Thus the OS needs to remember the **disk address** of a given **page**
- All of a process's memory is stored in the swap space in the form of pages. A subset of these pages are stored in main memory.
- If pages in memory are associated with the **code** segment, then the system can safely reuse these pages, knowing that it can later swap these pages again from the binary in **disk**

Present Bit

- Swapping is implemented via the **present bit**
- During a TLB miss, the hardware looks at the **page table** located in memory. If the PTE's present bit indicates that the page is not in memory (and only in disk), then the OS knows to copy the page from disk to memory (and also record it in the TLB)
    - The access of a page not in memory is called a **page fault**
    - The service that the OS provides in response to the page fault **exception** is called the **page-fault handler**

- A TLB hit implies that the page is already in memory. No swapping is needed

Page Fault

- An **exception** raised by the **hardware** which forces the OS to respond (via a handler).
- Occurs when the **present bit** is set to 0 when the hardware looks up a **PTE** in the **page table** (in main memory)
- The PTE also contains the **disk address** of the page we require from memory. When a **page fault** occurs, the hardware looks at the PTE and uses the disk address to find the page in disk
- Page faults are very slow. They require a disk I/O operation. Thus the overhead added by the OS (instead of having page faults handled by hardware) is acceptable
- When the disk I/O operation completes, the OS updates the page table. It may or may not also update the TLB cache. If it doesn't, then the system will encounter a TLB miss and have to update the TLB
- During the I/O operation, the process is **blocked**. Thus the OS will decide to context switch and run another process, which **overlaps** with the current I/O operation

Page Replacement Policy

- When **memory** is full, the OS's **page replacement policy** decides which page to replace. Picking an incorrect page can be very costly
- Instead of performing page replacement is full, the OS can use **low** and **high watermarks** (thresholds) to perform group page swapping. This can further increase performance

BEYOND PHYSICAL MEMORY: POLICY

Page Replacement Policy Depends on Memory Pressure

- **Memory pressure** is a measure of how much memory is free.
- When the OS has a lot of free memory in the **free list**, then a **page fault** would result in the OS assigning a free page from the free list to the faulting page
- When the OS doesn't have a lot of free memory, it has high **memory pressure**

Optimal Replacement Policy

- The optimal **page replacement policy** is one that replaces the page that will be replaced furthest in the future. This will result in the fewest possible cache misses. However, this is difficult to implement.
- The optimal policy serves as a good benchmark for policies attempting to approximate it

Types of Cache Misses

- **Cold-start miss**: Occurs when the cache is empty to begin with (ex: cache has been flushed)

- **Capacity miss**: Occurs when the cache is full (all pages have been allocated). It now has to evict a page and swap a page in
- **Conflict miss**: Occurs in **set-associative** caches, when there's a restriction to where something can be placed in the cache. Thus this does not occur in **fully-associative** caches such as the OS page cache

Policy: FIFO
- The first page to be accessed is the first page to be evicted.
- Easily implemented in a **queue** data structure

Policy: Random
- When under memory pressure, pick a random page to replace
- Does better than FIFO, worse than optimal

Policy: Least Recently Used (LRU)
- Uses memory reference history to determine the recently accessed page. Based on the **principle of locality**
- LRU ends up being more optimal than FIFO and Random, but is costly to implement. Implementation requires the moving of one element of the LRU queue data structure to the front of the queue (the MRU side)
- LRU is vulnerable to looping-sequential workloads (will always page fault). Random policy will do better here. To prevent this, LRU can use **scan resistance**

Policy: Approximating LRU with Clock Algorithm
- LRU is too expensive, let's approximate it instead with hardware support
- **Use bit** (a.k.a. Reference bit): A bit in each **page** that live in page table or another supporting structure.
- Whenever a page is referenced, the **hardware** sets the bit to 1
- The OS uses a **clock algorithm** operating on a circular list of all the physical pages of the system (and their associated **use bit**). It starts at any page on the list. When a **page fault** occurs and a **page replacement** is required, the OS checks if the current page of the algorithm has the use bit set to 1. If so, the OS sets it to 0, and repeats its check on the next page of the list, until it reaches a page that has a use bit set to 0.
- This **clock** approximation of LRU is very close to LRU

Clock Algorithm Considering Dirty Pages
- **Modified bit** (a.k.a. **Dirty bit**): a bit of a page that indicates if the page has been written to. A page that has the modified bit set to 1 is considered **dirty**
- Dirty pages are more expensive to evict than **clean** pages
- The clock algorithm can be modified to favor the eviction of clean pages

Page Selection Policies
- The OS must decide on a **policy** of when to bring a **page** into memory

- **Demand paging** is used for most pages, where pages are only brought into memory when they are referenced
- **Prefetching** is used to bring pages before memory is referenced when the system knows with good probability what pages are going to be required in the future

Write Page to Disk Policies

- The OS can decide to write individual pages to disk at a time
- **Clustering** is the collection of pending individual page writes to memory for one more efficient group write

Thashing

- When the memory demands of running processes exceed the capacity of physical memory, the system will constantly be **replacing pages** on every **memory reference**. This scenario is called **thrashing**
- **Admission control** is the system deciding not to run a subset of processes, in the hopes that attempting to do less work will be more optimal than trying to do everything at once
- Some OSes have **out-of-memory killers** that kill memory intensive processes when thrashing occurs

CONCURRENCY: INTRODUCTION

Thread

- An abstraction for a single running **process**. One process may have multiple **threads**. Thus we typically describe threads with respect to other threads of the same process they are a part of

Thread vs Process

- Each **thread** can be thought of as a separate process, except the fact that they all share the same **address space**, and thus may access the same data
- **Thread state** is similar to a **process state** in that each thread state has its own private set of registers (and thus their own PC)
- A **context switch** between threads is analogous to processes, as registers are saved to the **thread control block** (TCB). There is a TCB for each thread in the process
    - However, threads differ from processes during a context switch in that the **address space** does not change. Thus there is no need to change the per-process **page table**
- Although they share the same address space, each thread has a private **stack** called **thread-local** storage

Motivation for Threads

- **Parallelism**: Use a thread per CPU to do more work

- **Overlap**: Avoid the wait of a process blocking due to a slow I/O (ex: disk I/O, page fault). While one thread is **blocked**, the **scheduler** can switch to running another **thread** within the same program

Race Condition

- The results depend on the order of execution of code

Critical Section

- A piece of **code** that can result in a **race condition** if executed by multiple **threads**
- Code that accesses a **shared resource** (ex: shared variable) and therefore must not be run concurrently by multiple threads
- **Critical sections** require **mutual exclusion**, a guarantee that if one thread is executing it, no other thread is

Indeterminate

- A program is **indeterminate** if it has one or more **race conditions**

Atomicity

- Cannot be split further
- An **atomic instruction** is an instruction that is guaranteed to not be **interrupted** during execution
- The grouping of many actions into an **atomic** action is called a **transaction**

Synchronization Primitives

- **Hardware synchronization primitives** in combination with the **OS** allow

Condition Variables

- Useful for signalling between **threads**

CONCURRENCY: LOCKS

Problems in Concurrency

- We would like to execute a series of instructions **atomically**. But because of **interrupts** on a single processor or because of multiple **threads** executing on multiple processors we can't

Lock

- Addresses **concurrency** problems: are used to annotate **critical sections** of source code. Critical sections execute as if they were a single instruction
- **Locks** provide minimal control over **scheduling** to the programmer. **Threads** are entities created by the programmer but scheduled by the OS. Locks give back some of this control

Pthread Locks

- A POSIX library lock is called a **mutex**, as it provides **mutual exclusion** between **threads**

Coarse-grained Locking

- Instead of having one lock that protects multiple **critical sections** from multiple threads, it is more efficient to separate **critical sections** with different **locks**

Building a Lock

- A lock is evaluated by:
    - **Mutual exclusion**: It must prevent multiple **threads** from entering a **critical section**
    - **Fairness**: When the lock is released, all threads contending for the lock need a fair chance in acquiring the lock. Does any thread **starve**?
    - **Performance**: How fast is it in different conditions?

Controlling Interrupts

- An early implementation of locking was to turn off **interrupts**. That way, a thread can continue executing without the possibility of being interrupted and the system context switching to a different thread
- Pro: Simple to implement
- Con: Requires **privileged instructions** to turn on/off interrupts (and thus we must trust the user program)
- Con: Does not work on systems with **multiple processors** as disabling interrupts will not prevent other threads (on different processors) from entering a critical section
- Con: Turning off interrupts can lead to lost **interrupts**. (ex: How will the OS know how to wake the process waiting for a disk I/O)
- Con: Slow
- Due to these cons, **controlling interrupts** is seen typically only within OS code such as when accessing its own data structures. This is fine because OS is trusted

Spin-Waiting

- A **thread spin-waits** if it is waiting to acquire a **lock** (held by another thread)
- Very wasteful, executing instructions without doing anything

Evaluating Spin Locks

- Good correctness: they provide **mutual exclusion**
- Bad fairness: threads aren't guaranteed any fair chance of entering critical section
- Performance: bad on a single processor due to **spin-waiting**, might be okay for multiple cores.

Test and Set

- A **hardware primitive** that atomically tests for a value and sets a value at the same time
- Allows for the implementation of **spin lock**

Compare and Swap

- A **hardware primitive** that tests whether a value held by a pointer is what is expected. If so, set it to a new value. If not, then do nothing

- More powerful than **test and set** and can be used to implement **lock-free synchronization**

Load-Linked and Store-Conditional (MIPS)

- The **load-linked** is similar to **load** instruction in that it fetches a value from memory and places it into memory.
- The **store-conditional** is similar to **store** but only succeeds in storing a value if no intervening **store** has taken place

Fetch-And-Add

- **Fetch-and-add** instruction is a **hardware primitive** which automatically increments a value while returning the old value at a particular address
- Fetch-and-add is used to implement **ticket lock**, which uses a ticket and turn variable to build a lock. Ticket locks can be used to guarantee **fairness** since each contending thread has a fixed position in the queue and can't be cut in line

Yield

- A OS primitive (a system call) that a **thread** can call when it wants to give up the CPU and let another thread run
    - A thread can be **running**, **ready**, or **blocked** (similar to a process)
    - A **yield** changes thread state from running to ready. Thus a thread yielding is it **descheduling** itself (without needing the OS scheduler)
- Instead of having a thread **spin**, we can make it **yield** instead to increase performance
- While it is better than spin-waiting, **context-switches** can still be expensive
- Does not address **starvation**: a thread may endlessly yield without a guarantee to eventually run

Using Queues: Sleeping instead of Spinning

- The real problem is that if the scheduler makes a bad decision on which thread to run, it will either have to yield or spin. Either case, there can be **starvation** and high performance cost
- Given a way to make a thread sleep, we can implement a queue of sleeping threads that are woken up by the active thread when the lock is unreleased

Wakeup / Waiting Race

- A **race condition** in which thread A may try to wake up thread B before thread B has started waiting, which leads to thread B waiting indefinitely
- We can prevent **wakeup / waiting race** by implementing a way to make thread B not wait if thread A has already tried to wake thread B up (ex: setpark())

Two-Phase Locks

- **Spinning** can be useful if we know that the **lock** is about to be released
- In the first phase, the thread spins for a while, hoping to acquire the lock
- In the second phase, the thread **sleeps**, and only wakes up when the lock has been released

CONCURRENCY: CONDITION VARIABLES

Motivation

- In many cases, a thread wishes to check for a **condition** before continuing **execution**
    - Ex: Thread A may way to check if thread B has completed before executing
        - Often called join()
- We can use **condition variables** to replace **spin-waiting** with sleeping and waking up on a desired condition

Shared Variables

- A **shared variable**, a variable that both threads may access, can work. But it is inefficient, as threads can be spin-waiting while waiting for another thread to respond

Condition Variable (POSIX)

- A **condition variable** is an explicit **queue** that threads can put themselves in when a state of execution (some **condition**) is not as desired (**waiting** for **condition**)
- When a thread changes a relevant **condition**, it can wake up one or more threads in the **condition variable**
- A thread executes **wait()** to put itself to sleep
    - wait() takes in a **mutex** as a parameter
- A thread executes **signal()** when it has changed some of condition and it wants to wake up a sleeping thread that is waiting on this condition
- wait() and signal() can be used to implement join()
- **Always hold the lock** while signaling or waiting

Producer / Consumer (a.k.a. Bounded Buffer) **Problem**

- Producers produce items and place them on the **buffer** (bounded buffer). Consumers take from the buffer and consume these items. The **bounded buffer** is a **shared resource**
- Ex: 2 consumers, 1 producer. Consumer A sees empty buffer, decides to wait. Producer sees empty buffer, produces until full, signals to A. Consumer B sees non-empty buffer, consumes all of it. Consumer A wakes up expecting non-empty buffer. Error!
    - We want Consumer A to check the condition again. (use a while loop)
    - We want Consumer to signal the correct thread. (use different **condition variables** for consumer and producer)

- The problem is that we use **signalling** on a thread to **wake** it up because a **condition** has become desirable. But we have no guarantee that during its actual execution, the condition will stay desirable
- Another problem is that a thread may not wake up the correct thread. Recall that a thread wakes up another thread via signal(), which relies on a **queue** structure

Mesa Semantics
- The interpretation of what a **signal** means
- With **conditional variables**, always use while loops

Use While (not If) for Conditions
- Using while loops to check for **conditions** is always correct
- Using if statements to check for **conditions** is sometimes correct
- Using while loops also handles implementations of threads that cause **spurious wakeups**: where one signal may cause multiple threads to wake up

Covering Conditions
- If we're unsure of what threads to wake up, we may instead wish to wake up every thread associated with a **conditional variable**

CONCURRENCY: LOCK-BASED DATA STRUCTURES

CONCURRENCY: COMMON PROBLEMS

Deadlocks
- Occur when multiple threads wish to acquire locks (resources) held by other locks and cannot make progress because they are constantly waiting for each other

Conditions for Deadlocks (all must hold)
- The resource the threads wish to acquire must be **mutually exclusive** in that it can only be held by one thread or the other
- The threads that are holding the resources are **holding and waiting**: they wait to obtain more of the resource while holding the resource. That is to say, they don't try to perform a all or nothing operation
- **No preemption**: There is no way for a thread to be relinquished of its resources unless it does so naturally through acquiring the resources it needs and carrying out its operations
- **Circular wait**: There is a circular dependency of threads that each holds a resource that the next thread in the chain requires (cycle in a directed graph)

PERSISTENCE

Input / Output Device (I/O device)

- Hardware that receives data and sends data, typically via **bus**

System Architecture

- The structure of the system is hierarchical: devices with slower **I/O devices** are placed further away from the **CPU**. This is done because there is tradeoff between the length of a bus and its performance.
- The **CPU** and **memory** are placed close to each other, connected via **memory bus**
- Some devices such as **graphics** and other high performance **I/O devices** are further away and connected via **PCI bus**
- The slowest devices such as **mice**, **keyboard**, and **disk** are connected by a **peripheral bus** (such as USB) and are the further away from the **CPU**

Lowering Overhead of Programmed I/O (CPU overhead)

- We can reduce the overhead of **polling** with **interrupts**
- We can reduce the overhead of the CPU writing to the I/O device with **DMA**

Polling vs Interrupts in Devices

- When the OS wants to use an **I/O device**, it needs a way to know when the device is available. The OS can **poll** the device, but this is inefficient as we spend a lot of CPU time checking on whether if the device is available (**Programmed I/O**). It is more efficient to have the device send an **interrupt** to the CPU when it is available.
- Using interrupts instead of polling means that the CPU can spend its extra time on another process instead of spin waiting
- However, we should use polling instead of interrupts when we expect that the device will be available very soon (**context switching** is expensive)
- Sometimes, we use a **hybrid** of interrupts and polling when the speed of a device is unknown.
- **Coalescing** can be used to combine multiple interrupts into one interrupt which reduces overall overhead. The tradeoff is higher **latency**

Direct Memory Access (DMA)

- Orchestrates data transfer between **I/O devices** and **memory** with little CPU intervention
- The CPU tells the **DMA engine** where to buffer data in **memory**, which **device** to send data to, and how much data to transfer
- The DMA engine can then perform the I/O operation while the CPU is free to do something else

Interacting with Devices

- There are two primary methods: using extra **instructions** or using extra **registers**

- We can use explicit **I/O instructions** (privileged) that specify how the OS sends data to devices
- We can use **memory mapped I/O**: use the existing store and load instructions to read and write data on **registers** reserved for devices
- No great advantage from using either

Device Driver (part of the OS)
- The specific interface of devices are handled by software in the OS called **device drivers** that layer between the device hardware and the application
- Device drivers are the OS's way of **encapsulating** low level details of devices and making the rest of the OS device neutral

HARD DISK DRIVES

Hard Disk Drive (HDD)
- An **I/O device** that serves as a main form of **persistent storage**

HDD Interface
- An HDD consists of 512 byte blocks named **sectors**. We can view the HDD as an array of sectors, with 0 to n-1 (n being number of sectors) as its **address space**
- **File systems** may perform multi-sector operations such as 4KB reads/writes, but HDD manufacturers only guarantee **atomicity** to single sectors
    - Power failures may result in **torn writes**
- Prefer **sequential access**, where accessing blocks that are closely together sequentially is typically faster than random access

Physical Components of a HDD
- **Platter** is a circular hard surface where data is stored using magnetic charges
    - A disk may have one or more platters
    - Each platter has two sides, called **surfaces**
- Platters are bound together around a **spindle** which is connected to a motor that spins the platters
- **Sectors** are stored in concentric circles called **tracks** lying on the **surfaces**
- Reading and writing is performed by the **disk head**
    - There is one disk head per surface
- **Rotational delay** is the latency of having the head wait on the rotation of a track to read/write
- **Seek time** is the latency of having the **disk arm** move the **disk head** from the current track to the correct track to service a read/write
- **Transfer time** is the latency of having the head doing the actual read/write
- Modern HDDs have some **memory** acting as their **cache** which store data read/written on tracks

- On writes, HDDs can choose to **report immediately** by **writing back** once its memory has been written, or they may choose to **write through** once the disk itself has been written. The former is faster, but can be more dangerous (what if: power failure?)

Disk Scheduling

- I/O operations are expensive. The OS has a **disk scheduler** that decides which I/O operation shall be performed based on **shortest job first** (greedy algorithm). It can do this because I/O operations can be accurately approximated
- **Shortest seek time first**: Early algorithm where the queue of I/O requests was ordered by picking the nearest tracks. However, this is hard to implement as the OS sees an array of sectors, not concentric tracks
- **Nearest block first**: Instead of looking at shortest seek times, OS can schedule by nearest block address. However, this causes **starvation**

Elevator (a.k.a. **SCAN**)

- Disk scheduling algorithm that has the head move back and forth across all tracks, servicing requests in order along those tracks. A single pass across the entire disk is called a **sweep**
- SCAN is not effective in reducing **rotational latency**

Shortest Positioning Time First

- Disk scheduling is determined by accounting for rotational and seek time and choosing the best option (while conforming to the **elevator** algorithm)

Where is Scheduling Performed?

- On modern systems, the OS tries to choose an optimal set of I/O requests. This set is optimized further by a scheduler internal to the HDD
    - Implies that I/O operations are not done in the order OS thinks it is

I/O merging

- The OS merges I/O operations on consecutive blocks into one operation, which reduces the overhead of requesting the I/O

Anticipatory Disk Scheduling

- It can be more efficient for the HDD to wait for a certain amount of I/O requests instead of serving requests immediately

RAID

Redundant Arrays of Inexpensive Disks (RAIDs)

- Uses multiple **HDDs** to build a disk system that is faster, larger, and more reliable
- Externally, a **RAID** appears as a disk

- Internally, a **RAID** consists of: multiple disks, volatile and non-volatile memory, and processors that manage the disk system
    - Similar to a computer system that manages a group of disks

RAID advantages over a single disk

- Performance: Using multiple disks in parallel speeds up I/O
- Capacity
- Reliability: Using multiple disks for backups

Fail-Stop Fault Model

- This **fault model** has disks either being working or failed

RAID Level 0: Striping

- Perform **striping**: spreading the blocks of the array across all the disks of the RAID in a **round robin** fashion
- This results in parallel I/Os, maximum capacity, and no reliability (every bad sector results in data loss)

RAID Level 1: Mirroring

- Extend RAID to also keep duplicate disks
- Reads can be performed on just one disk, while writes must be performed on all disks of the same duplicate group. However, they can do so in parallel
- **Mirroring** with two disks halves the performance of **sequential reads and writes**.
    - Read performance is halved because mirrored disks are skipping over sectors that another mirrored disk serviced without providing useful throughput
    - Write performance is halved because two disks need to be written for one piece of information
- **Mirroring** with two disks achieves optimal performance of **random reads**
    - All the disks of a mirrored group can perform different reads
- **Mirroring** with two disks is halves the performance of **random writes**
    - Even with randomness, both disks still need to be written, halving throughput

RAID Level 4: Parity

- Extend RAID by **increasing capacity** while **decreasing performance**
- Dedicate a disk to be a **parity disk**: each block of this disk is determined by XORing its associated data blocks on other disks. The parity disk can check for bad sectors by comparing its value to new XORs
- Sequential and random reads can be spread across all non-parity disks (N-1)
- Sequential writes can take advantage of **full stripe writes**, which parallelizes each disk in a XOR and write operation across all disks that take the time of one disk write
- Random write time is bottlenecked by the I/O access of the parity disk. When using **subtractive parity** to perform these writes, random write time is R/2, where R is speed of a 1 disk write

RAID Level 5: Rotational Parity

- Rotates the parity blocks across all disks, which improves **random write time** to (N/4) * R
    - There are 4 I/Os in this case: a read and write to the parity disk, a read and write to the updating data disk

Which RAID to Use?

- Use RAID Level 1 Striping for performance, capacity
- Use RAID Level 2 Mirroring for performance, reliability
- Use RAID Level 5 Rotational Parity for capacity, reliability

FILES AND DIRECTORIES (UNIX)

File

- A **file** is a linear array of bytes associated with a low level name called an **inode number**

Directory

- A **directory** is a **file** (and thus has its own **inode number**) that is content specific: contains a list of user readable names associated with low level names (**inode numbers**)
- **Directory hierarchies** are built by placing directories inside directories

File Descriptor

- A private per-process integer used to access a file (read/write) if it has the right permissions
- File descriptors are a **capability** because having it gives you the ability to perform certain operations

Metadata API

- Applications use the stat(), fstat() system calls to examine a file's metadata
- CLI provides the stat command to examine a file's metadata

Inode

- Contains **metadata** on a **file**
- A persistent data structure residing on disk, with some copied onto memory for caching

Creating Hard Links

- We can create a reference to the same file using ln with the CLI and the link() system call

- Creating a **hard link** means creating an entry in the directory: this entry is composed of a name and associated inode number. Thus multiple files can reference the same piece of data if their **inode numbers** are the same

Creating Files

- The creation of a file is equivalent to the creation of an **inode** data structure which tracks all relevant information on the file and associating it with a name that the user can recognize. This name is shown in the directory

Deleting Files

- There can be multiple files referencing the same inode
- The **file system** keeps track of how many links there are between an inode and its existing files
- Deleting a file means unlinking the file name with its inode data structure. Once an inode has no more file associated with it, its memory is truly free

Symbolic Links

- Creating a **hard link** to a directory runs the risk of creating a cycle in the directory hierarchy. Thus we only allow hard links to non-directories
- A **symbolic link** holds the pathname of the file that it is referencing. When we create a symbolic link file to a file X, the symbolic link's inode number is not the same as X. If we had created a hard link, they would share the same inode number.
    - This implies that a soft link can be left **dangling** if file X is renamed or removed because the only information that the soft link has in accessing the data that X points to is the name of X itself

Access Control Lists

- A more general way of defining who may access files than using **permission bits**

Creating a File System

- We create a file system by associating a **disk device** (such as /dev/sda1), a **file system type** (such as ext3), and the directory of the current directory tree that we want to mount the root of the new file system on
    - Ex: mount -t ext3 /dev/sda1 /home/users
    - In this ex, an unmounted file system associated with /dev/sda1 is mounted to the the directory /home/users
- **Mounting** allows for the unification of file systems into a single directory tree

FILE SYSTEM IMPLEMENTATION

- **File systems** are implemented completely with software

File System Organization

- The **file system** views the disk as an array of **blocks** of fixed size
- Most blocks are **data blocks**
- There are blocks dedicated for storing **inodes**, called the **inode table**. These contain all metadata on files and may point to **data blocks**
- There are blocks dedicated to managing allocation/deallocation of **data** and inodes**.** These are the **data block** and **inode bitmaps**. Each bit in the bitmap represent the freeness of a block or an inode
- There is a **super block** that the **OS** uses to know the type of **file system**, position of bitmaps, position of inode table, and position of data blocks

Approaches in Managing Data Blocks

- **Multi-Level Index:** Each **inode** contains data block pointers. These can be direct, indirect, doubly indirect, or triply indirect. Direct pointers contain the address of data blocks. Indirect pointers point to other pointers, thus magnifying the amount of blocks an inode can manage
    - Up to 3 extra I/Os
- **Extents:** An extent is a pointer to disk and an associated length, which describes the amount of contiguous data it is pointing to
- **Linked Based** (FAT file system)**:** No inodes. Directories contain one pointer to the first data block. Each data block contains a pointer to the next data block associated with the file. Since this is slow, these pointers are stored in a table in memory. But this takes up a lot of RAM
    - Up to n extra I/Os
    - Bad for supporting **hard links**

Reads Don't Access Allocation Structures

- Data block and inode bitmaps are not accessed during reads because all the information required is in the other structures

Reading A File From Disk

- Reading a file means traversing from / to the filename. The number of I/Os in a read is proportional to how many directories we traverse (the cost of **opening**)
    - Every level is 2 I/O's: one to read the inode, one to read the data of that inode (to find the next directory)
- Starting from root (at predetermined inode number 2), we search for the next part of the pathname in the / directory and find its associated **inode number**. We use it to find the next inode and repeat until we've found the file we wish to read

Writing To Disk

- To write, we first incur the cost of **opening** the file. Unoptimized, each write takes 5 additional I/Os: reading and writing both bitmaps, and writing to the data block

Buffering and Caching

- We can **cache** memory so that reads need less I/O
- We can **buffer** (collect) a bunch of writes and combine it into one large I/O, so that we need less I/O

Disk Fragmentation

- As the disk ages and accrues external fragmentation, new files will have their data scattered throughout the disk in non contiguous chunks, which reduces access time dramatically
- **Defragmentation** is the process of reorganizing file data to be contiguous

Organizing Structure: The Cylinder Group

- Physically, a **cylinder** is a set of tracks on different **surfaces**. Faster access speeds can be obtained when we keep accessing the same cylinder
- A **cylinder group** is a single **cylinder**
- In the **file system**, cylinders are organized as **block groups**. Accessing the same block group repeatedly results in short seek times

Block Groups

- Each **block group** has its own **super block**, **inode bitmap**, **data block bitmap**, **inode table**, and **data blocks**
    - The bitmaps keep track of the inodes and data blocks within the block group

Policy: How to allocate files and directories

- To reduce access times, the file system attempts to keep a file's data within the same block group (because this translates to lower seek time as we're using one cylinder group). It also keeps unrelated data apart from each other. If we do this, we create more space to data that is more relevant to fit in the right block group
- **Directory Placement** (FFS): Find a block group with low directory count, high inode count. This will balance directories across block groups and also make sure we have a lot of inodes to assign other files in that directory to
- **File Placement** (FFS): Data blocks are associated with an inode. Place data blocks within the same block group that their associated inode is also in. Thus we prevent long seeks as an access to a file is within the same cylinder group
- This directory + file placement approach means that files in the same directory will often have all their data contained with the same block group, increasing speed further when we expect the user to access a directory's files repeatedly
    - We are exploiting **name-based locality**: we expect repeated access to the same directory
- **Large File Exception:** A large file with all of its data in one block group is bad because other local files cannot fit with it in the same block group. It's better to spread a large file's data across many block groups. The seek cost is relatively low compared to the transfer of data (**amortization**) and we preserve the advantages of **locality**

Handling Small Blocks

- **Sub-blocks** are units smaller than blocks that hold data for small files. Enough of them can be combined into a single **block**
    - This is I/O costly, due to needing to copy over sub-blocks into blocks
- We can speed this up by having the OS buffer writes before issuing single block writes to the file system that include all the data of the sub-blocks

Parametrization

- Typically, it isn't efficient to allocate a file's data actually contiguously. The disk head moves too quickly for the system to actually perform a reads on contiguous blocks.
- Instead, the **file system** can **parametrize** the allocation of blocks to be in a different order (such as every other or every 3rd) depending on the performance of the disk

Track Buffer

- The cost of **parametrization** is the fact that every skipped block not read is a block wasted. Thus we can **buffer** the track after its been read once, and read from cache (memory) once we need it again

CRASH CONSISTENCY: FSCK AND JOURNALING

- **File systems** must **persist** and preserve data even when the system crashes (due to power failure or an OS bug)

Crash Scenarios (for writing data onto disk (which requires updating data bitmap, inode, data))

- Suppose a crash happens right after only one of these happen:
- **Updated data block**: This is fine. It will appear as if the write didn't happen
- **Updated inode:** The inode now has a pointer to **garbage data**. There is inconsistency: data block bitmap says block isn't allocated; inode says block is allocated
- **Updated bitmap**: Inconsistency: data block bitmap reports allocation, but inode doesn't know its allocated. This leads to **space leak**: file system will never use this block
- **Updated bitmap and inode**: Metadata is consistent, but the file has garbage data
- **Updated inode and data block**: Inconsistency between bitmap and inode
- **Updated bitmap and data block**: Inconsistency: we have no idea which inode the data belongs to

Solution 1: File System Checker (**fsck**)

- Tool for finding metadata inconsistencies (between inode and bitmap)
    - It cannot fix the case when the metadata is consistent but data block is not
- **Superblock**: Checks to see if superblock seems reasonable
- **Free blocks**: Scan the inodes, data blocks, and indirect blocks. When there is a metadata inconsistency, **fsck** assumes that the inodes are correct
- **Inode state**: Checks inode for corruption (ex: invalid field). Suspect inodes are cleared and the inode bitmap is updated

- **Inode links**: **fsck** builds its own directory tree from traversing the directories. It can count the number of links (from counting files with same inodes) and compare that with the inode link count in the inode. Typically, an inconsistency is resolved by changing the inode's link count
    - If an allocated inode is found without a directory enclosing it, it is moved to the **lost+found** directory
- **Duplicates**: If two inodes are pointing to the same data block, we can remove a corrupted inode or create copies of the data for both inodes
- **Bad blocks**: checks blocks for bad pointers (pointing outside partition range)
- **Directory checks**: ensure "." and ".." are first entries, inodes referred to by directories are allocated, and directories are linked at most once in the directory hierarchy
- **fsck** works but is inefficient

Solution 2: Journaling (a.k.a **Write-Ahead Logging**)

- A **journal** is **non-volatile** memory that is written ahead of actual writes to disk. It is implemented as a circular buffer
- We create a **transaction**, where we write a start transaction start block and the data all at once to the **journal**
    - If we crash here, then it is as if the data was never written
- After that is done, we can **atomically** write a transaction end block to mark the transaction in the **journal**
    - If we crash here, no progress is lost
- Then we can write the transaction to disk in a process called **checkpointing**
- Some time later, mark the transaction **free** by updating the journal super block

Metadata Journaling

- If we were to journal by making a copy of physical writes, then there would be 2x traffic for writes
- Instead of **data journaling**, we can perform **metadata journaling** where all data except for the data blocks themselves are recorded in a journal
- This type of journal requires the writing of data first before the journaling of metadata. This is because the data isn't recorded in the journal, and thus we risk having the **inodes** store **garbage** data if we have the data be written after the journaling process (during a crash)

1. Write Data
2. Journal Metadata
3. Wait for 1. And 2. Journal commit by completing the transaction with an end transaction block
4. Wait for 3. Perform checkpointing (do the actual write (of the metadata) to disk the transaction promised to do)
5. Free up the transaction in the journal for future writes

Tricky Case: Block Reuse (Overwriting data)

- Suppose you write X, delete X, then write Y. Suppose they share the same data block
    - When we write X, we also writes its content to the log.
    - Suppose we crash after writing Y's data (not recorded in journal)
    - When we reboot, we read the journal and load X's data into the block (and thus overwriting Y's data). We end with Y's metadata for X's data
- We can prevent this by having the journal record **revokes** when data is deleted: this will prevent X's data from being written when we replay the journal

LOG-STRUCTURED FILE SYSTEM

Log Structured File System

- When writing to disk, the **LFS** first buffers the writes into a memory **segment**. Instead of writing to the location that the user asks to write (thus no overwriting data), LFS **sequentially** transfers the data into a **free** part of disk
    - Because these writes are large and sequential, we take advantage of **RAID** in avoiding the **small-write** (random writing) case in which we get I/O bottlenecks from writing
- The goal of the **LFS** is to turn all writes into **sequential writes**, because then they take advantage of **full stripe** writes of a **RAID**
    - Since the I/O typically come from from writes, we favor paying the cost of needing to **random read** in favor of **sequentially writing**
- These writes include **data** and **metadata**

Write Buffering

- Since the time it takes the disk to rotate from one block to another is not the same as the speed at which we can issue writes to a block, we need to instead **buffer** our writes and combine them into one large write before we issue the one large write to disk
    - This one large write that LFS eventually sends to disk is called a **segment**

Using Indirection to Implement LFS

- Each **sequence** has an adjacent **inode map** that points to the inodes of the sequence, which in turn point to the data we want to read.
- To find these **inode maps**, we need a reserved region in the beginning of our disk (so that we always know how to find it) called the **checkpoint region** that contains the addresses of all **inode maps**
- After a period of time, the system **caches** all of the **inode map** by reading the **checkpoint region**. It can then access the inodes and data with the same number of I/Os as a typically UNIX FS (both have immediate access to a given inode)

LFS Solves the Recursive Update Problem

- Suppose you update inode X of a file. With LFS, we don't overwrite the inode (writing it as the same location). Instead, we write the updated inode to free memory. But this

means that the directory that contains information to the address of inode X is incorrect, and must be updated as well. But then this means that the directory that contains this directory has bad information and must also be updated => **recursive update problem**
- LFS solves this problem through the **inode map**: instead of updating the directory, we update the inode map. Thus for our new segment, the new directory is associated with the correct data

LFS Garbage Collection
- LFS works on a **segment** granularity in freeing up memory. It combines segments with some dead blocks and combines them into segments full of live blocks. It frees up the old segments and writes the segments full of live blocks
- Problem: How does LFS know where the live blocks are?

MECHANISM: Determining Block Liveness
- LFS can determine if a block is live by checking its **inode** number. Using the **inode map**, it can find the inode and see where the inode thinks the block is. If there is a match, then the block is live. Else, the block is dead

POLICY: Which Blocks to Clean?
- One policy is to clean **cold segments** first, where the segment is rarely being updated and to wait before cleaning hot segments that are constantly being updated

Crash Recovery and the Log
- To handle writing to **checkpoint region** correctly: LFS keeps two CRs and updates them in an alternating fashion with timestamps between the actual updates
    - This results in a copy of the CR that has consistent timestamps
- To handle writing to **disk** correctly: LFS looks at the last segment pointed to by the CR. If it can find neighboring segments that aren't in the CR, it will **roll forward** and add their locations to the CR

EXT2 FILESYSTEM

Block Device
- A computer storage **device** that supports the reading and (optionally) writing of data in fixed size **blocks**
- Ex: hard disk drive (HDD), solid-state drive (SSD)

Block
- The fixed unit of storage of data that the **block device** and **file system** manages
- The use of block sizes instead of word sizes for **I/O** can be

Block Group

- A collection of blocks close to each other, which reduces **seek** overhead and reduces **external fragmentation**
- Information on every **block group** is kept in the **descriptor table** that is located right after the **super block**
- The two blocks near the start of each **block group** are the **block usage bitmap** and **inode usage bitmap**. These blocks show which blocks and inodes are being used by the enclosing **block group**
- Since each **bitmap** is limited to the size of a block, the max size of a block group is 8x the size of a block
- The blocks after the bitmaps in the block group contain the **inode table** and the remainder are **data blocks**

Directory
- A **file system** object. It has an **inode** just like a **file**
- It is a specially formatted **file** that containing data that associates a **file**'s **name** with an **inode number**
- In modern Ext2, **directories** also contain the **type** of the object (file, directory, device, socket, etc)

Inode (index node)
- Each **object** in the **filesystem** is represented by a data structure called an **inode**
- An **inode** contains the locations of all **data blocks** of an object (using pointers)
- An inode contains all **metadata** of an object, except for the object's name
- All inodes are stored in **inode tables**. There is one **inode table** per **block group**

Superblock
- Contains configuration information about the filesystem: # of inodes, # of free inodes, when filesystem was mounted, whether file system was cleanly unmounted, when it was modified, filesystem version, OS that created it
- Stored after the **boot block**, with backup copies scattered around many other blocks

Symbolic Link (a.k.a. Symlink / Soft Link)
- A special **file** that contains a reference to another **file** via a **pathname**
- If the referenced file (the target) is moved, then the **symbolic link** is **dangling**
- Since symbolic links are **files**, they have their own **inodes**
    - Inodes may store the symbolic link data directly if it is is small enough
- Symbolic links can point to the files of other **devices** and **file systems**

Disk Organization
- **Ext2 filesystems** start with a **superblock**

Device Major and Minor Numbers

- Devices with the same **major number** use the same **device driver**
- When accessing a **device file**, the **minor number** is used as an argument. It is left to the device driver to implement the response to the access

DATA INTEGRITY AND PROTECTION

Latent Sector Errors (LSE)

- Errors in the disk that are **easily detected**
- In a **RAID-4/5** system, we can just use a **hot spare** to reconstruct the failing disk
    - We can also use multiple parity disks to mitigate the risk further at the cost of decreasing usable space

Block Corruption

- **Block corruption** is difficult to detect. They can result because of writes to wrong location, faulty hardware (such as a bad bus), etc. They cause **silent faults**

The Checksum

- To detect **block corruption**, we use **checksums**, which is the result to a piece of data when we apply a function to it
- We want different checksums for data that is not the same, and the same checksums for data that is not the same
- Ex: XOR checksum, addition checksum, Fletcher checksum, **cyclic redundancy checksum**: divide data D by agreed upon value k

Checksum Layout

- Checksums can be placed right next to the blocks they check (8 bytes + 512 bytes) if the disk is made that way
    - This makes writing sequential (when writing to a block): write to checksum and block for 1 I/O operation
- Another layout: n checksums can be placed before n blocks (for drives that only support 512 bytes), but this is costly in writes: write to checksum, write to block

Using Checksums

- When reading a block D, we also read its associated **checksum** C. We perform operations on D that results in another checksum C'. If C and C' do not match, we think that D is corrupted

Misdirected Write

- A write that is successful (I/O wise) but writes to the wrong location

- We can add a **physical identifier** (physical ID) during our write that specifies where the written data should reside. Thus whenever we read (and thus we know where we are reading), we can check that the physical ID matches that location we wish to read. If no match, there is corruption

Lost Writes

- A **lost write** occurs when a device informs the upper layer that the write has occur but the write doesn't **persist**
    - What remains is the old contents of the block that the upper layer thinks the device has written on
- Solution 1: **Verify after write**: read the contents after a write attempt to check that we've actually written
    - This doubles the number of I/Os however
- Solution 2: **Write checksum**: have a checksum that can check if the write persists (with the hope that the write checksum was successfully written)

Scrubbing

- The system can scan every block and its checksum to check for corruption every so often in a process called **disk scrubbing**

Checksum Overhead

- Space overhead: occupies space in disk for each block, occupies space in memory if we wish to check quickly
- Time overhead: CPU must compute the checksum for each block whenever it is written (to find the new checksum) and whenever it is read (computing checksum to compare with other checksum)
    - We can reduce this by combining data copying and checksumming into one faster operation (than doing both separately)

FLASH-BASED SSD

Solid-state storage

- **Persistent** storage with no moving parts and is built from transistors

Challenges of Flash

- Expensive to erase data (have to erase a lot at once)
- Repeated overwrites to the same location eventually wears out the storage

Organization

- **Cells** (single-level, double-level, etc) store bits (1,2,3)
- **Banks** store cells
- A bank can be accessed via a **erase block** granularity: (128 - 256 bytes)

- An **erase block** contains **pages**: (4 bytes)
- To write to a **page** within a **block**, we must first **erase** that entire block before we can write to it

Flash Operations

- Read (a **page**): can read a **page** from anywhere on disk at equal speed (**random access**)
- Erase (a **block**): before we can write to a **page**, we must erase the entire **block** that the page is within. This means setting all bits to 1 (thus we lose information during erase)
- Program (a **page)**: We can change some bits to 0. This is effectively writing a page

Disturbance

- When reading or programming a **page**, it's possible to change the bits of other pages
- To minimize **disturbance**, **FTL** will typically program pages in an order from low page to high page in an erased block

Flash Translation Layer (FTL)

- **FTL** is the interface that the filesystem uses to access the SSD device
- FTL takes read and write requests on **logical blocks** and translates these requests into actual reads, writes, and erases on the **physical pages** and **physical blocks** of the SSD device

Achieving Better Performance

- We can **parallelize** the flash drive into multiple **flash chips**
- We wish to reduce **write amplification** (the overhead of actually writing per write request)

Direct Mapping

- Logical page N is directly mapped to physical page N
- This is expensive: whenever we wish to write onto a part of a logical page, we must save a copy of the physical block of the page, erase the block, and then program the new and old pages
- This is prone to **wear out**: there may be bias towards overwriting particular data

Log Structured FTL

- Instead of directly mapping logical pages to physical pages, we always write data to free pages (just like in a LFS)
- We keep an **in-memory mapping table** to keep track of block addresses
- This makes writing data much faster as we have to erase less often and we are accruing the erase/write cost of overwriting data

Garbage Collection for Log Structured FTL

- Since writes are log style, we need **garbage collection** to remove blocks that have some **dead pages** and copy live pages over to the next block
- We can use the in-memory **mapping table** and extra information associated with each page to find out which page is **dead**: Read what **logical page** the physical page is associated with. Using the **mapping table**, map the logical page to the actual physical address the logical page is stored in. If the physical page addresses do not match, then we've found a dead page

Overprovisioning

- **Garbage collection** can be costly. Some SSDs are **overprovisioned** with extra flash capacity, so that garbage collection is less likely to be needed for space during active use. When the device is idle, garbage collection can then happen => great performance

Mapping Table Size

- An **in-memory mapping table** that maps logical pages to physical pages is too large to be stored in RAM
    - Instead, we can use **erase block** granularity
- This will reduce the amount of memory we need for our mapping table, but will greatly increase the cost of **small writes**
    - Overwriting a **logical page** will then mean having to write an entire **physical block**, because our mapping table is working with a block granularity

Hybrid Mapping

- To reduce memory (when using block pointers in the mapping table) and avoiding large writes (when using page pointers in the mapping table), we can decide to use both
    - The page pointers are stored in the **log table**, while the block pointers are stored in the **data table**
- This will benefit us in memory and fast small writes, but will complicate **garbage collection**

Hybrid Mapping Garbage Collection

- The goal is to merge page pointers into block pointers when we can
- This is easy when we sequentially overwrite a logical block. We write each physical page with its address to the log. Once we have overwritten all the logical pages, we can have a block pointer point to the block containing the new physical pages, instead of page pointers
    - This is **switch merge**
- **Partial merge**: If we don't have all of the logical pages of a block overwritten but need to free memory anyway, we can logically overwrite them and force a merge. This is more expensive
- **Full merge**: If we don't have all of the logical pages of a block and those logical pages are stored in other blocks, we have to perform many writes to not only free the original block, but all other blocks that contain the pages that we need to merge it

Caching

- We can get even faster performance by caching flash data into memory
- Risks include needing to constantly **evict** pages (we cannot contain the **working set** in memory) and needing to write **dirty** pages during eviction

Wear Leveling

- In order to prevent blocks from **wearing out**, we use **wear leveling** to spread out erasing/programming. Thus all blocks should wear out at the same time
- The log structured FTL does a good job of **wear leveling**
    - However, sometimes data is never written to certain blocks. This means that wear leveling is not being utilized for these blocks that are never erased
- To improve on log structured FTL, we periodically write all live blocks (now in different locations) to get better **wear leveling**

Trim

- The **trim** operation can inform the SSD device when blocks are no longer used, which makes **garbage collection** more efficient

DISTRIBUTED SYSTEMS

Distributed System

- A system that tolerates failure of individual components and uses its components to provide a better service than a single computer can

Communication is Unreliable

- Sending data over a network is prone to failure as packets may be tampered with or **dropped** at a network switch, out of the hands of the communicating systems

Unreliable Communication Layers

- Instead of trying to achieve reliable communication, we can have our applications deal with **unreliable communication**
- **UDP/IP** is a **network stack** that has unreliable communication, but features detection for data **corruption** through its **checksum**: data is sent along with its checksum; when the data is received, its checksum is computed and compared with the arriving checksum

Reliable Communication Layers

- Reliable connections feature the receiver sending **acknowledgements** to the sender. If the sender never gets the acknowledgement, it will **timeout** and **retry** by repeating the sending of the message
- If the **acknowledgement** is lost, the sender thinks that the message was never sent even though the receiver received it. If the sender retries, it will send the same message twice (BAD)
- We can use **counters** that both sides agree on. The sender increments its counter whenever it sends a message along with its counter value. The receiver expects the message to match the receivers counter value. If it does, the receiver increments its counter when it sends an acknowledgement.
    - If an acknowledgement is lost, the sender will retry and the receiver will know that the message is a duplicate

Distributed Shared Memory (DSM)

- Multiple machines share a **virtual address space** and have their own physical **address spaces**. Note that we're using an OS abstraction of memory to build a distributed system
    - This means that when a machine wants to access a piece of memory, it will either be in its own local physical address, or in another machine's physical address. It will **page fault** and ask another machine for the data
- This system is unreliable, as the failure of a machine can mean the failure of the system (what if that machine contains vital information that will be lost)
    - Thus no one uses **DSM** today

Remote Procedure Calls (RPC)

- Objective: make the process of executing code on a remote machine be as simple as calling a function

- **Stub generator**: automates the conversion from the function call to the message needing to be sent over the network. There is a **client stub generator** and a **server stub generator**
- Typically better to use a RPC on an **unreliable communication layer** to utilize performance. If it were to use a reliable one, it would effectively be sending extra messages. We can implement reliability within the RPC layer instead of using the communication layer
- RPC handles communication between machines of different **endianness** by converting messages to a different endian data format when necessary
- Typical RPCs are **synchronous**: when the client issues the RPC, it has to wait for continuing its execution. But this wait can be long, and so some RPCs are **asynchronous**: the client can do other work instead of waiting

Thread Pool

- Instead of waiting for requests and serving them one at a time, a server can use a **thread pool** of worker threads and one thread that checks for requests to take advantage of **concurrency** (we can be doing work and also checking for requests at the same time)

RPC Fragmentation and Reassembly

- The **stub generators** are responsible for breaking large data into messages and then putting them back together

Fallacies of Distributed Computing

1. There is no latency
2. Bandwidth is infinite
3. There is one administrator
4. The network is secure
5. Communication is reliable
6. Transport costs are free (inside the system)
7. Topology doesn't change (such as nodes of the network)
8. Network is homogenous (machines have same word sizes, architecture, etc)

AUTHENTICATION

Principal

- The party asking permission to do something
- Once the **principal** is **authenticated**, the system will usually rely on the first authentication (and thus not spend time checking later)

Agent

- The computing entity that requests the **object** on behalf of the **principal**

Credential

- Information that the OS uses to determine if a user has the permission to access an **object**

Methods of Authentication

- What you know
- What you have
- Where you are

Passwords

- Systems store the **hashes** of passwords, not the actual plain-text passwords themselves. Hashes are non-invertible, and thus there is less danger of knowing a password given its hash
- In addition to **cryptographic hashing**, systems often use a **salt**: a random number added to the actual password. This way, the same password on multiple sites have different hashes because the sites used different **salts**. This negates attackers using dictionary hashes to match stolen hashes (stolen passwords)

ACCESS CONTROL

- Suppose **subject** X wants to access an **object**. It might or might not be allowable. Now what?

Access Control List

- Analogous to having a doorman check the person's name before letting him enter
- If we make the ACL too long, we get large overheads in accessing it to check permissions and storing it.
- The **UNIX ACL** uses 9 bits, 3 groups, and 3 modes of access to allow for a short ACL that gave a descent range of access

Capabilities

- Analogous to giving permissible people a key to the door
- Since **capabilities** are at the low level bits, they run the risk of being forged or being copied. Because of this danger, only the OS maintains the capability. Processes can perform operations on **capabilities**, but must do so by having the OS be its intermediary
- Implementation: have a pointer in the **PCB** (process control block) point to a list of the process's **capabilities**
    - This can incur a high overhead
- Advantages: checking a principal's ability to access objects is easy. Giving a subset of capabilities to a child process is easy, while its difficult to subset permissions for ACL

- Disadvantages: for a given file, knowing who can access it is difficult compared to a ACL because we must check each principal instead of checking the ACL

Cryptographic Capabilities

- Since having the OS store **capabilities** is expensive, we can instead have the user applications store it, and use a hashing algorithm to determine if its correct

Role Based Access Control (RBAC)

- Designate principals as being in a particular role, and give them access to objects associated with their role
- They require an authentication step to enter a new role, and the relinquishment of privileges associated with the previous role

PYTHON

Special Methods (a.k.a. Magic Methods)

- **Special Methods** are **methods** of a **class** that execute in specific circumstances
- Ex: __init__, __str__, __repr__
- Ex: To define a class's constructor, we define the class's __init__ method

Python Functions

- The **parameters** of a Python **function definition** are called **formal parameters**
- The **arguments** of a Python **function call** are either **positional arguments** or **keyword arguments**
- Functions in Python are **neither pass by value or pass by reference**
    - The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object). [1] When a function calls another function, a new local symbol table is created for that call.
- If no return value or no return statement is used, a function always returns **None**
- **Function definitions** support **default argument values**

    - Ex: `def ask_ok(prompt, retries=4, reminder='Please try again!'):`
- **Function calls** support **keyword arguments** (ex: ask_ok(prompt="hi"))
    - Keyword arguments must follow positional arguments
-

KEYWORDS / OPERATORS

in

- Tests whether or not a sequence contains a particular value
- Ex:
```
ok = input(prompt)
if ok in ('y', 'ye', 'yes'):
    return True
```

==

- Tests whether or not two operands have the same value

is (analogous to === in PHP)

- Tests whether or not two operands refer to the same object

DATA TYPES

Iterator

- A class that defines the __next__ method

Iterable

- Any object (not necessarily a data structure) that defines the __iter__ method which returns an **iterator** and the __next__ method
- An **iterable** can have __iter__ return itself, which makes itself an **iterator**
- Ex: **containers** such as lists, sets, dictionaries

list

- Initialization: my_list = [1,2,3]
- Add elements: my_list.append(4)

dict (dictionary, the standard mapping type)

- A **mapping** object that maps **hashable** values to arbitrary objects
- Mappings are **mutable**

MODULES

Package

- A collection of python **modules**

Module

- A Python **module** is a single python file that generally contains **variables**, **functions**, and **classes**
- Can contain **statements** and **function definitions**, where statements are executed when a module is initialized using **import**
- Contains a private **symbol table**, which is used as the global symbol table by all functions defined in itself.
    - This avoids name conflicts between modules.
    - We can touch a module's global variables: modname.itemname
- Can import other modules using **import**
- Ex:
  import fibo.fib #imports the fibo.fib module name into local symbol table
  from fibo import fib #import module fibo.fib as fib in the local symbol table
  import fibo.fib as fibonacci #imports fibo.fib into the local symbol table, fibonacci is a reference
- Modules are **imported once per interpreter session** and are reloaded by: `importlib.reload(modulename)`

Executing Modules as Scripts

- A module can act as both a script as well as an importable module.
    - if __name__ is "__main__" can be used to execute code when the module is called directly: `python fibo.py <arguments>`
      If the Python interpreter is running that module (the source file) as the main program, it sets the special `__name__` variable to have a value `"__main__"`

Module Search Path

- The interpreter searches for imported modules first in built-in modules, then in variable **sys.path**, a list of strings
- A directory that contains **packages** must include the __init__.py file in order to be put in the module search path

Standard Modules

- Python comes with a library of standard modules, called the **Python Library Reference**
- The variable sear

virtualenv module

- Module that supports python **virtual environments**
- Create a virtual environment: python3 -m venv ~/python_environments/djangodev

    - Specify the interpreter and directory to contain environment
- Activate a source: `source ~/python_environments/djangodev/bin/activate`
- When a virtual environment is activated, pip will install modules in that virtual environment and not to the global site packages
- A **python virtual environment** contains an instance of a python interpreter and modules that will be run with the interpreter.
    - Ex: virtual environment A contains: python 3.7 binary, module X with version N

DECORATORS

- Any Python **callable** object that is used to **modify a function or class**. It takes in a function or class as argument and returns a modified function or class
- Two kinds of **decorators**: **function decorators** and **class decorators**

DJANGO

- Python **framework** for **web development**
- Used in **server backend** through the Python language

Creating a Django Project

- `django-admin startproject mysite`
- Avoid naming projects after python modules or Django components
- In web development with PHP (with no modern framework), code is stored in the web server's **document root** (ex: /var/www). In Django, code is stored outside of the document root for security
- **startproject** command creates files:
  mysite/ #container for the project
    manage.py #lets you interact with the project
    mysite/ #python **package** name
      __init__.py #specifies that current directory is a **package**
      settings.py #settings for project
      urls.py #URL declarations for project
      wsgi.py #an entry point for WSGI compatible web servers to serve project

Django Development Server

- Start the server: `python manage.py runserver`
- The default server IP address is: http://127.0.0.1:8000/
- Run with different IP: python manage.py runserver 8080

- The development server automatically reloads Python code for each request as needed. You don't need to restart the server for code changes to take effect. However, some actions like adding files don't trigger a restart, so you'll have to restart the server in these cases.

Django Project vs Django Application

- A web app does something. A project is a set of web apps and configuration files for a website. A project can contain multiple web apps. A web app can be in multiple projects

Creating a Django Application

- `python manage.py startapp polls #creates an app called polls`
- The resulting directory structure houses the application:

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

Views

- A python function that **takes in a web request** and **returns a web response**
- We map **views** to **URL**s in order to associate a request on an URL to a corresponding response by the server

URLconf (URL configuration)

- User defined python module which maps between **URL path expressions** and **views**

django.urls.path(route, view, kwargs=None, name=None)

- **route** is a string that contains a URL pattern
- **view** is the handler
- **kwargs** is keyword args
- **name** is unique identifier that can be used in any part of Django

include()

- Used to include URL patterns
- `admin.site.urls` is the only exception to this.

How Django processes a HTTP request

1. Determines the root URLconf module to use
2. Loads that module and locates the urlpatterns, which is a list of django.url.path()
3. Runs through each URL pattern, in order, and stops at the first that matches the requested URL
4. Import the **view** associated with the URL. The view is passed an instance of HttpRequest,
5.

Migrations

- **Migrations** are derived from **models**. They allow for the version control of a **database schema**
- Migrating is creating tables for the database:
  - `$ python manage.py migrate`
- Storing migrations (to migrate later):
  - `$ python manage.py makemigrations polls`
- See the SQL code that will be run when you migrate migrations:
  - `python manage.py sqlmigrate polls 0001`

Models (Django's mapping between a Python class and a database table)

- A **model** maps to a **database table**
- **Models** in Django are Python **classes** (subclassing a generic Django Model class) that Django maps to a **database**
  - Each class **model** has variables that map to a database **field**
- Example of writing a model:

```
class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
```

- Question is a class (model)/table, CharField is a class variable/field, question_text is a variable instance/column name (left is Python/right is database)
- **Models** are part of **Django apps** and can be implicitly linked when Django apps are included in **Django projects**

Motivation for Models and Migrations

- A Django **user** creates a **model** so that Django can create a **database schema** for him

1. The user creates **models** via Python (and the Django model API)
2. The user runs **makemigrations**, which stores instances of his model that can be converted to SQL code
3. The user runs **sqlmigrations**, which outputs the SQL code that running **migrations** will execute, resulting in the **database schema**
4. The user run **migrations**, which runs migrations (runs the SQL code that creates the database schema)

Including a Django Application to a Django Project

- Add a reference to the configuration class in project

Database setup

- By default, Django uses **SQLite** as the RDBMS for the **project**

Template Engine

- Allows for static template files in the application
- During **runtime**, the server replaces variables in the template file with actual values and transforms the template file into an HTML file to be handed out to the client

Templates Organization

- Instead of placing templates in one location (like we do with **static files**), we place templates according to the **app** it is for
- Templates that are **app specific** should be placed in project/app/templates/app/template_file
- Templates that are **not app specific** (such as custom admin templates) should be placed in project/templates/template_file
- Placing templates with respect to if they are app specific or not makes the **app** easier to be reused for future **projects**

Generic Views

- A **generic view** is an abstraction for common types of web responses
- Some **generic views** need to know what **model** they are acting on via the model attribute
    - For example, the **DetailView** uses this information to know which object it is displaying the details of, and what predefined template name to use

Generic View Examples

- ListView abstracts the concept of displaying a list objects
- DetailView abstracts the concept of displaying a detail page for a particular type of object

Django Tests

- Run tests:
    - $ python manage.py test polls
- Create tests by defining classes in tests.py of the particular Django app. These classes extend the Django **TestCase** by defining methods (for assertions) with names starting with “test

Client Tests

- We can use django.tests.client to interact with our site at the view level
- We do this by having the client make requests and examining the response (similar to using a browser)

Static File Organization

- **Static files** (CSS, Javascript, image files) can be scattered across multiple **Django apps**
- django.contrib.staticfiles is used to get files from a single location

Packaging an App

- **Python packaging** refers to preparing an app to where it can be easily installed and used

Forms

Form

- In the context of **HTML**, forms are a collection of elements inside the tags <form>...</form> that **allow users to send information to the server**
    - **GET** and **POST** are the only **HTTP methods** that handle forms
- In the context of **Django**'s role in forms,

Django Form Classes

- **Django Forms** create forms

Building a Form

-

C / C++

KEYWORDS

Global Variable

- A **global variable** is a **variable** that has been defined outside of any function. Their scope starts at the point of **definition** to the end of the program. They can be **externally linked** to other source files, which means the same name refers to the same location in memory

static

- A **static variable** is a **variable** that has been allocated **statically** at compile time: its lifetime is the entire run of the program. It is stored in the **data** segment of memory

volatile

- A **qualifier** applied to a **variable** during **declaration**
- A variable should declared **volatile** whenever its value can be changed unexpectedly:

1. Memory-mapped peripheral registers
2. Global variables modified by **interrupts**
3. Global variables accessed by multiple tasks within a multi-threaded app

I/O

Stream

- A sequence of bytes

Buffer

- An area of **memory** allocated for the temporary storage of data

Reading from stdin and writing to stdout using library functions

```
while((curChar = (char)getchar()) != EOF){
      mappedChar = map[curChar];
      putchar(mappedChar);
   }
```

Reading from stdin and writing to stdout using system calls

```
while(read(0, buf, 1) > 0){
      //map the read in char, and output into stdout
      buf[0] = map[buf[0]];
      write(1, buf, 1);
   }
```

```
char* mallocChar(void){
   return (char*)malloc(1);
}
```

Handle

- A **handle** is a function that is called when a **signal** or **event** occurs
- Usage: signal(sig, handle), where sig is the signal and handle is the function to call when sig occurs.
- Ex: Suppose sig = SIGTERM. Call signal(sig, handle). When the process receives the signal SIGTERM (such as from the terminal), the process will execute handle

Wait

- wait, waitpid - wait for a child process to stop or terminate

int dup2(int oldfd, int newfd)

- The process's file associated with newfd is replaced by the file associated with oldfd. If the file associated with newfd was open, it will be closed.

SQL

SQL (Structured Query Language)

- SQL is a standard **language** for accessing and manipulating **databases**
- There are different versions of SQL that all follow **ANSI** standard along with additional proprietary features
    - Following ANSI standard means all versions of provide interfaces for: SELECT, UPDATE, DELETE, INSERT, WHERE
- SQL is used to **query** a **RDBMS** database
- SQL is **not case-sensitive**

RDBMS (Relational Database Management System)

- **SQL** queries a **RDBMS**
- **RDBMS** is a **DBMS** (often implemented in C) that uses a **relational model** of data. (i.e. It is a computer running a program that allows for SQL queries)
- Ex: MySQL, SQLite are RDBMSes

Table

- A **table** is a database object that is a collection of **related data entries** and consists of columns and rows
- A table is broken into smaller entities called **fields**

Field

- A **column** in a **table** that is designed to maintain a specific information about every **record** in the table

Record

- A **row** in a **table**

Important SQL Commands

- **SELECT** - extracts data from a database
- **UPDATE** - updates data in a database
- **DELETE** - deletes data from a database
- **INSERT INTO** - inserts new data into a database
- **CREATE DATABASE** - creates a new database
- **ALTER DATABASE** - modifies a database
- **CREATE TABLE** - creates a new table
- **ALTER TABLE** - modifies a table
- **DROP TABLE** - deletes a table
- **CREATE INDEX** - creates an index (search key)

- **DROP INDEX** - deletes an index

SELECT [field]
- SELECT selects data from a **table**
- The result is stored in a table called a **result-set**

SELECT DISTINCT [field]
- Similar to SELECT, but the **result-set** will only contain unique **records**

WHERE [condition]
- Filters records that follow conditions
- **AND**, **OR**, **NOT** can be used to combine conditions
- **IS NULL**, **IS NOT NULL** are operators used to evaluate conditions involving **NULL**
- **LIKE** [pattern] filters out records that match a pattern
- **IN** [(value1, ... , valueN)] specifies multiple values for WHERE
    - Ex: WHERE col1 IN (val1, val2)
- **BETWEEN** [value1] **AND** [value2] specifies a certain range
-

ORDER BY [field]
- Order records by a values of a **field**
- Ex: SELECT * FROM table ORDER BY field DESC/ASC
    - Select all fields from table. Sort by field in desc/asc order

INSERT INTO [table]
- Insert new records to the table
- The first way specifies both column names and values to insert:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

- If data for a column is omitted, then the value returns **NULL**
    - **Auto Incrementing** fields automatically scale as records are inserted

UPDATE [table]
- Update existing records of a table
- Update a table by setting all values that

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- If WHERE clause omitted, then condition is always true

DELETE [table]
- Delete entire **records** from a table that follow a condition:

```
DELETE FROM table_name
WHERE condition;
```

Wildcard Characters

- **%** is 0 or more characters
- **_** is 1 character
- [charset] is 1 character is in charset
- Ex: The **pattern** [!abc][def]% matches strings that don't have a, b, or c as first character, have d, e, or f in second character.

Aliases

- Create a temporary reference to a **table** or **field**
- `SELECT column_name AS alias_name`
- Aliases exist for the duration of the **query**

ALTER TABLE [table]

- Used to **ADD** columns, **DROP** columns, or **MODIFY** a column's data type
- Ex:

```
ALTER TABLE table_name
ADD column_name datatype;
```

Primary Key

- A primary key is a **field** where each **record** has a unique, non-null value

Foreign Key

- A **field** in one table that refers to the **primary key** of another table
- Typically, a foreign key must be the same data type as a primary key
- Adding a foreign key to a table means adding the primary key of a related table. We should **normalize** both tables before doing so, so that we don't modify foreign keys after we add them

Relationship

- Multiple **tables** containing related data have a **relationship**

Database Schema

- The logical skeleton of the database

Transaction

- A all or nothing update to the database.
- Can be seen as an **atomic operation** that

NORMALIZATION

Normalization

- Process of organizing data in a database
- Goal is to **protect data**, **eliminate redundancy**, and **inconsistent dependency**

Redundant Data

- Wastes space
- If data is changed, it must be changed everywhere

Inconsistent Dependency

- When a **field** can be related to other fields in a **table**, the field is **dependent** on data in that table
- When a **field** seems to be unrelated to other fields in a **table**, the field is **inconsistently dependent** on that table
- We wish to avoid **inconsistent dependencies** as they naturally result in **redundant data** and confusion

Normal Form

- **Normal forms** are rules for database normalization
- Third normal form is the highest level necessary for most applications

First Normal Form

- Eliminate repeating **groups** in individual tables
    - Ex: Class 1, Class 2, Class 3 in a Student table
        - This can be seen as **one-to-many** relation. We should place one-to-many relating data in separate tables
- Create a separate table for a set of **related data**
- Identify each table with a **primary key**

Second Normal Form

- Create tables of sets of values that apply to multiple records
- Relate these tables with a **foreign key**
    - The idea is that a **field** should be uniquely stored in one table dedicated for it. When we wish to relate information from this table to another table, we use a **foreign key**

Third Normal Form

- Eliminate **fields** that do not depend on the key
    - If a **field** can be included in multiple tables, consider creating a separate table that contains that field, and relates it with other fields with a **foreign key**
- It may be more practical to avoid third normal form

DATABASE RELATIONSHIPS

Database Relationship

- Relationships between **tables** of the **database**

One-to-one (mapping in both directions is injective)

- Record A in table 1 is only related to record B in table 2. Likewise, record B in table 2 is only related to record A in table 1
- Tables with a **one-to-one** relationship usually can be combined into one table without breaking **normalization**

One-to-many (mapping in one direction is non-injective)

- Record A in table 1 is related to any number of records in table 2. The records in table 2 that are related to record A in table 1 are not related to any other records in table 1
- We can relate a table that has a **one-to-many** relationship with another table by adding a **foreign key** (primary key from table on “many” side) to the table on the “one” side

Many-to-many (mapping in both directions is non-injective)

- Record A in table 1 is related to any number of records in table 2. Record B in table 2 is related to any number of records in table 2
- Tables with a **many-to-many** relationship usually require a third **linking** table, because **relational** systems can’t directly accommodate this relationship

SQLITE

Creating a Table

- CREATE TABLE table_name(field_name data_type, PRIMARY KEY / NOT NULL)

MACHINE LEARNING

Supervised ML

- Combine inputs
- Produce predictions
- Able to do so on never before seen data

TERMINOLOGY

Label

- Y variable in regression
- Labels are what we're predicting. (y variable). It is the y variable in basic linear regression.

Features

- X variable in regression
- Features are our inputs. They're the ways we represent our data. They are the variables xi in basic linear regression.

Example

- An example is a unit of data. An example is a particular instance of the variable x.
    - **Labeled Example**
        - An instance of a pair (features, label) = (x,y), where both components are
        - A labeled example is a set of features that has been labelled with the answer. It is a pair (features, answer). (features are our inputs, answer is our output)
    - **Unlabeled Example**
        - An instance of a features
        - An unlabeled example is a set of features that has

Model

- Models maps examples to predicted labels y'
- Models are defined by internal parameters, which are learned
- Two main phases of what is a model goes through is **training** and **inference**

Training

- Show the model labeled examples and make it learn the relationships between the label and the features
- Training a model means giving a model a set of labeled examples, and examining many examples

Inference

- Have the model predict useful values from unlabeled examples.

Regression vs Classification

- Regression models handle continuous values. (What is the value of this house?)
- Classification models handle discrete values. (Is this email spam or not spam)

Linear Regression Model

- A linear regression model is a linear map that maps features to labels.
- Example: 1 label, 1 feature. Our model is y' = wx + b, where y' is our predicted label, x is our feature, w is the weight for this feature, and b is the bias for all features.
- More complicated models with more weights may look like y' = b + w1x1 + w2x2 + w3x3

Defining L2 Loss Function for Regression

- L2 Loss for a given example is also known as squared error.
- L2 Loss = (sum over all labeled examples{ (predicted y - labeled y)^2 }) / |D|
    - Where D is set of examples

Relationship between Linear Regression Model and L2 Loss Function

- We calculate Linear Regression