1. Operating Systems & Networks

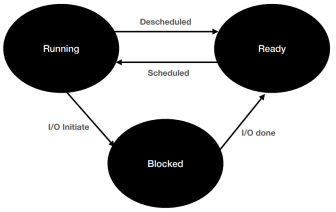
1.1 Introduction

1.1.1 Operating System

- ➔ Software that acts as an **intermediary between computer hardware and user apps**
- ➔ Manages computer's hardware resources (CPU, memory) and I/O devices (printers)
- ➔ Enables user programs to execute without worrying about hardware specifications.
- ➔ Three pillars of OS :
  - **Virtualisation** : Providing illusion of infinite memory and compute (CPU)
  - **Concurrency** : Running multiple processes at a time
  - **Persistence** : Managing storage on disk (hardware) using file systems (software)

1.1.2 Process

- ➔ A **program** is nothing but code; and executing programs are called **processes**.
- ➔ A process constitutes of a unique identifier (Process ID), memory image (static (code and data) and dynamic (stack and heap)), CPU context (registers, instruction pointer and program counter) and file descriptors (pointers to open files and devices for memory I/O).
- ➔ A process can be in one of the following **three states** :
  - **Blocked** : Waiting for some I/O call (not ready to run)
  - **Ready** : Waiting to be executed (ready to run)
  - **Running** : Executing on processor (running)



- ➔ Steps to **create a process** :
  - Load program into memory (lazy load from disk)
  - Runtime stack allocation (used for local variables, function parameters and return arguments)
  - Creation of program heap (used for dynamically allocated data)
  - Basic file setup (for STDIN, OUT, ERR)
  - Initialise CPU registers (setting PC to the first instruction)
  - Start the program

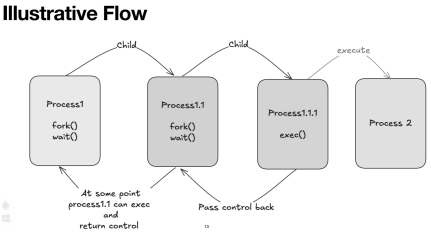
Time	Process 0	Process 1	What's happening
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process 0 initiates I/O
4	Blocked	Running	Process 0 is blocked, 1 runs
5	Blocked	Running	
6	Blocked	Running	I/O of process 0 is done
7	Ready	Running	Process 1 is done
8	Running	-	Process 0 is done

- ➔ OS uses process list to store metadata about processes, called **Process Control Block (PCB)**. It includes Process ID (identifier), process state and address space of the process (the register values).
- ➔ **init** process is the ancestor of all processes.

1.1.3 System Calls

- ➔ OS-provided function that allows user programs to interact with hardware.
- ➔ Two modes of execution : **User** and **Kernel** (higher privilege such as I/O)
  - Uses **Limited Direct Execution (LDE)** as a low level mechanism to separate user space from kernel space.
  - Kernel performs system calls on behalf of the user process. Uses a separate kernel stack and **Interrupt Descriptor Table** (IDT, aka Trap Table) to keep logs of different kernel functions addresses.
- ➔ **TRAP Instruction** : Special instruction to switch from user to kernel mode.
  - CPU to higher privilege level, save context (old PC, registers) on Kernel Stack, look up in IDT and jump to trap handler function in OS code.
  - Once done, the OS calls a special return-from-trap instruction which returns into the calling program, and back to user mode.
  - Different from interrupt (which are signals sent to CPU due to unexpected events, from either software or hardware), as it is a purely software generated interrupt caused by system calls or exceptions.

- ➔ **POSIX (Portable Operating Systems Interface)** : Standard set of system calls that an OS must implement to ensure portability. Programming languages abstract systems calls. Eg: printf() in C internally invokes write system call.
- ➔ Some important system calls :
  - **fork()** : Creates a new child process (with new PID), image copy of parent with independent memory. The new process is added to the list of processes and scheduled; and both start execution just after fork (with different return values).
  - **exec()** : Used to load a different executable to the memory of a child process and make it run a different program from the parent. We can also pass an executable in some variations.
  - **wait()** : Puts the parent in block state until the child terminates (options like waitpid() also exist). It also collects exit status of the terminated child process, providing some visibility to the parent process. wait allows the OS to reclaim the resources of the child and prevent **zombie processes**. init process adopts / reaps orphans.
  - **exit()** : Terminates a process



1.2 Process Virtualisation

- ➔ Multiple processes (even more than the number of processors) need to be executed with each of them having the illusion of exclusive access to resources.
- ➔ Requires a switching mechanism (hardware) along with some policies (software)

1.2.1 Switching

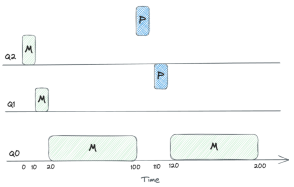
- ➔ **Context Switch** : A low-level piece of assembly code that saves a register value from executing process registers to kernel stack and restore values for the next process. Essentially return-from-trap will go to the new process.
- ➔ For switching (kicking the currently executing process), we have two approaches :
  - **Cooperative / Non-Preemptive** : OS trusts the processes to behave reasonably (Give control back - yield() call).
    - In case of a misbehaving process (eg: trying to do something they shouldn't), trap instruction transfers to the OS which terminates the process.
    - Problem : Reboot system, in cases of infinite execution of some process
  - **Non-Cooperative / Preemptive** : OS takes control with the help of interrupts.
    - Every X milliseconds, raise an interrupt -> halt the process -> invoke interrupt handler -> OS regains control (continue with the process or switch)

1.2.2 Scheduling

- ➔ Coming to scheduling policies, let's first define some **metrics** :
  - **Performance** : $T_{turnaround} = T_{completion} - T_{arrival}$ *and* $T_{response} = T_{firstrun} - T_{arrival}$
  - **Fairness** : Jains fairness index (fairness in scheduling)
- ➔ For I/O jobs, the scheduler simply moves the job to blocked state. Once I/O is done, an interrupt is raised and the OS moves the process back to ready state.
- ➔ Some assumptions, which we'll start with : All jobs arrive at the same time, Each job runs for the same amount of time, No I/O times for any job and Known run times.
- ➔ Various scheduling policies are as follows :
  - **First Come First Serve Policy** : Schedule the job that came first. As soon as it is done, schedule the job that comes next. Good when all above assumptions are held. But, if processes take different times, might lead to convoy effect (longest process hoarding and not letting smaller processes finish fast).
  - **Shortest Job First (SJF) Policy** : Assumes that all jobs come at the same time, and prioritises ones which will finish fast.
  - **Shortest Time to Completion First (STCF)** : Advanced SJF that does not assume jobs come at same time and switches whenever a faster completing job enters.
  - **Round Robin Scheduling** : Tries improving on the response time by running jobs for time slices (Run job for a time slice → switch to next job → while true) instead of individual job to completion. Time slice duration is also important, too

small wastes a lot of time on context switch; and too large leads to CPU hogging

- **Multi Level Feedback Queues (MLFQ)** : Assigns processes to different priority queues. If priority(A) > priority(B), A runs; if equal, they share CPU (round robin). Priorities are adjusted based on behavior, promoting interactive jobs and demoting long-running CPU-heavy ones. This reduces turnaround and response times but may lead to starvation (solved via periodic priority boosts) and gaming (processes faking I/O to stay high priority).



  - Determining periodic boost interval is hard (voo-doo constant). Too small might not give proper share to interactive jobs; and too big might starve long running jobs.

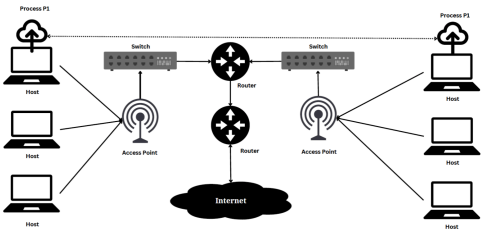
1.3 Networking

➔ Client sends a request to the server, to which the server provides a response.

➔ **Protocol** : Agreement between communicating parties on how to communicate

➔ Software components in the OS that support (using system calls) network calls are called **protocol stack**. Few hardware components :

- **Host** : Any device that send or receive traffic (can be client or server)
- **IP Address** : 32 bits, hierarchically assigned address used by host to send data
- **Repeater** : Allows regeneration of signals for long distance communication
- **Hub** : Multi-port repeaters (Key issue: everyone receives everyone's data)
- **Bridge** : Sits b/w two hubs and is aware about hosts on either side (for routing)
- **Switch** : Multi-port bridge. Devices connected to a switch are part of 1 network
- **Router** : Facilitate communication between networks. Act as traffic control point facilitating security, filtering and redirection.
  - All communications to / from out of the network goes through the router.



➔ Multiple types of networks exist :

- **Personal Area Network (PAN)** : Very short range. eg bluetooth (master-slave)
- **Local Area Network (LAN)** : Private network operating within / nearby a single building. Wireless LANs : 11 Mbps to 7 Gbps, wired LANs 100 Mbps to 40 Gbps. One large Physical LAN can be divided into smaller logical LANs (Virtual LANs)
- **Metropolitan Area Network (MAN)** : City wide networks
- **Wide Area Network (WAN)** : Span large geographical areas (country, continent, etc). Higher latency and lower transmission speeds. Internet is a large WAN (Dedicated WANs for large organisations also exist, costly tho)

➔ Networks are often organised as a stack of layers for abstraction. The set of layers along with protocols forms the **Network Architecture**.

- Layer n of one machine communicates with layer n of another using a protocol.
- Between each pair of adjacent layer there is an interface, which defines the primitive operations and services the lower layer makes available

1.3.1 The OSI Model

➔ Open System Interconnection (OSI), a conceptual framework used to understand how network communication works through different layers.

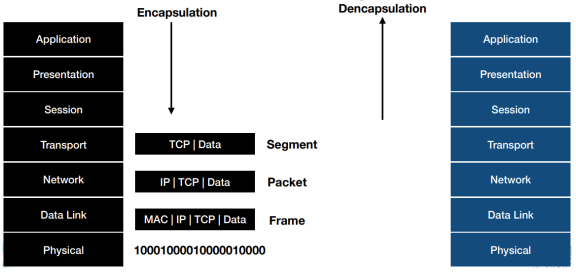
➔ Facilitate interoperability between different technologies

➔ Comprised on 7 layers :

- **Physical Layer (L1)** : Transmission of raw bits through physical medium. Comprises ethernet cables, optical fiber, coaxial cable, WiFi, hub, repeater, etc.
- **Data Link Layer (L2)** : Interacts with the physical medium using MAC address (12 hex digits). Ensures hop-to-hop communication by creating reliable links (error-correcting) between two directly connected (physically adjacent) nodes. Comprises NIC, WiFi access cards and switches (move data).
- **Network Layer (L3)** : Manages end-to-end communication (routing through different routes in a large network) using IP addressing (4 octets in IPv4).

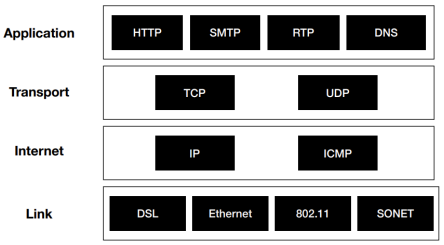
Performs logical addressing (IP), path selection and packet forwarding. Comprises routers, hosts, L3 switches, etc.

- **Transport Layer (L4)** : Service-to-service communication, ensuring that the right process receives the (reliable, sequential and free from others) data. Manages flow control and error correction. Uses ports (16-bit : 0-65535, privileged: 0-1023, registered: 1024 - 49151) to send / receive data, unique to each process. Comprises TCP and UDP.
- **Session Layer (L5)** : Manages (establish, maintain and terminate) connection between different devices.
- **Presentation Layer (L6)** : Data encryption & compression to ensure that data is in format that sender / receiver can understand.
- **Application Layer (L7)** : Provides support for end applications to format and manage data. In turn they make use of transport layer protocols. Comprises HTTP, DNS, SMTP, etc.



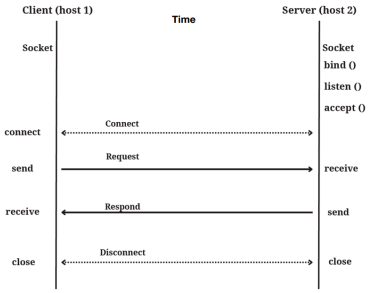
➔ We also have another model **Internet Model (TCP/IP Model)**, which, unlike OSI, is an actual practically used model with 4 layers :

- Application Layer : Corresponds to application, presentation and session
- Transport Layer : Transport layer of OSI
- Internet Layer : Network layer of OSI
- Network Layer : Physical and data link layers of OSI



1.3.2 Transport Layer

➔ **Socket API** : Simple abstraction, allowing applications to attach to the network at different ports. Socket establishing calls (like connect, accept, etc) are blocking calls, ie, raise trap instruction.



➔ Application process is identified by tuple (IP, Protocol, Port)

➔ We have many types of links :

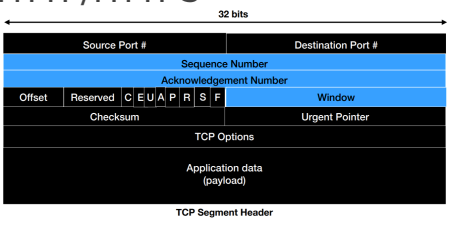
- **Full-duplex** : Bidirectional (both way at the same time)
- **Half-duplex** : Bidirectional (only one-way at a time)
- **Simplex** : Unidirectional

➔ Two step process :

- **Multiplexing** (sender) : Handle data from multiple sockets, add transport header
- **Demultiplexing** (receiver) : Use header info to deliver received segments to correct socket

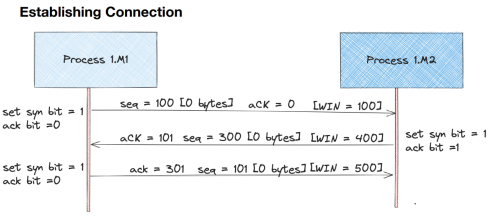
➔ Two types of protocols :

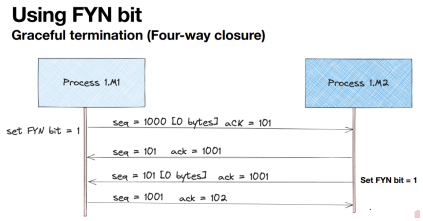
- **User Datagram Protocol (UDP)** : Opposite of TCP (like not connection oriented, no flow control, retransmission, etc). Use case : VoIP, DNS queries, streaming.
  - UDP socket identified using destination IP and port. UDP segments with the same destination port are redirected to the same socket.
  - UDP segment header (32bit wide) comprises of source and destination port (16bit each) in the first row, length (in bytes, including header) and checksum (of UDP, header, payload and pseudo header from IP layer) in second row, followed by the application data (payload).
- **Transmission Control Protocol (TCP)** : Connection oriented (establishing connection before transmission), congestion control, reliable (order maintained, error detection using acknowledgement and retransmission), higher overhead (~20 bytes > ~8 bytes), and flow control (adjust transmission rate or message limit based on network). Use case : mail, file transfer, HTTP/HTTPS
  - TCP sockets are identified using source IP & port and destination IP port. A server can support multiple TCP sockets, each communicating with a different client.
  - A TCP packet comprises of sequence number

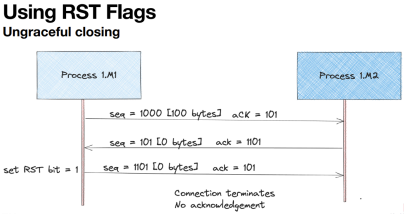


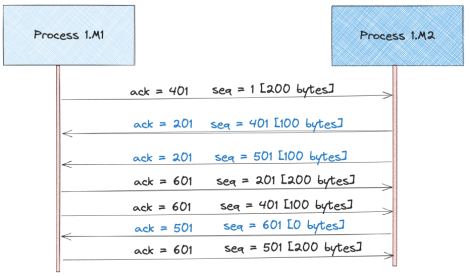
TCP Segment Header

(no. of sent bytes), acknowledgement number (next expected byte seq number), window (no. of bytes the receiver can accept), A (acknowledgement bit), R & S & F (connection management), C & E (congestion notification) and offset (length of the TCP header).



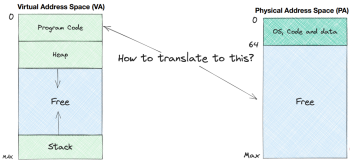






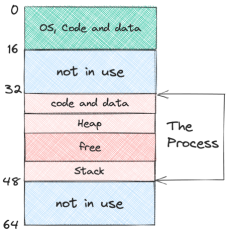
1.4 Memory Virtualisation

- ➔ Goal : Illusion that each process has its own private memory, while in reality, many processes share the same memory. Transparency (user prog abstraction), efficiency (min overhead) and protection (dedicated & isolated spaces for each process).
- ➔ **Address Space** : Comprises program code (and static data), heap (dynamic memory allocations) and stack (function calls during runtime). OS allocates memory and tracks the location of the process.
  - ◆ Static/global variables are allocated in executable, stack memory (aka automatic memory) allocations and deallocations are managed implicitly by compilers and heap memory is handled explicitly by the programmer.
  - ◆ brk and sbrk are a few memory management system calls that increase / decrease the size of heap based on value. malloc(), free(), mmap(), calloc(), etc are built upon these. Modern programming languages support these implicitly.
  - ◆ CPU loads / stores to a virtual address (VA) but memory hardware needs to access physical address (PA). This address translation is done by the Memory Management Unit (MMU) - hardware.
- ➔ To achieve virtualisation, we need some hardware support for execution and OS mechanisms to control and manage. Three Key assumptions:
  - ◆ Address space contiguously placed in physical memory
  - ◆ Size of address space is less than size of physical memory
  - ◆ Each address space is of exactly the same size

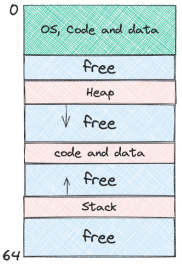


1.4.1 Memory Management

- ➔ **Base and Bounds** : Each process allocated contiguous memory (segment). Two hardware registers in the MMU : base register and bounds/limits register. Each program is written and compiled as if it is loaded at 0. However, during execution, the OS decides the location in physical memory and sets the base register to that value.
  - ◆ Hardware calculates physical address as virtual address (process generated) + base; at runtime (dynamic relocation)
  - ◆ Limits : One base-bounds register pair per process, no large address space support (any address access beyond bounds lead to interrupt and process termination), lots of wasted space between stack and heap (internal fragmentation), handling more memory demands than bound by a process.



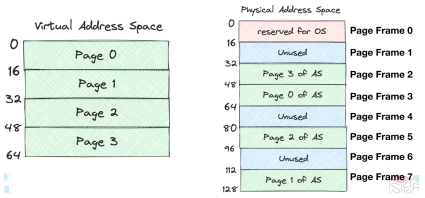
- ➔ **Segmentation** (Generalised base-&-bound) : Instead of one base-bound per process, have it per logical segment (code, segment and heap), allowing each segment to be placed in different parts of memory. The registers for storing these values are called segment registers.
  - ◆ Handles large and sparse address space well.
  - ◆ VA uses 14 bits, the first 2 identify the segment (00 : code, 01 : heal, 11 : stack, 10 : invalid) and the remaining 12 provide the offset.
    - Some systems consider code and heap as one segment and use only one bit.
    - Another method to identify the segment is using address formation



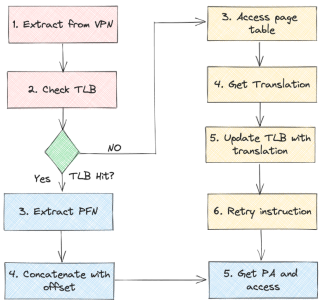
(program counter generated : code, stack pointer : stack, otherwise : heap)

- For address translation, get offset in VA for the specific segment and offset the same from the PA segment's base (note that stack grows up, unlike the other 2).
- Same issue of saving and restoring segment registers for each VA. Results in a lot of little holes across the physical address (external fragmentation).
  - These holes can be used with the help of memory tracking and algos like best-fit, first-fit, buddy algorithm, etc.
  - Another solution (when space larger than individual holes is needed) could be to stop all processes and cluster the empty spaces (memory intensive). We can only minimise, not avoid.

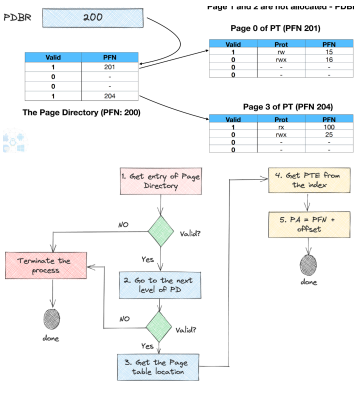
➔ **Paging** : Split the address space into fixed sized units, called pages. It's all about mapping page in VA to page frame in PA.



- The OS uses page tables, a per-process data structure, for address translation from virtual page number to page frame number.
  - Page tables can be huge, thus, stored in-memory.
  - Each entry is called a page table entry (PTE). Each PTE consists of a number of bits (apart from VPN and PFN) :
    - Valid : unused pages are marked invalid, access to which results in trap
    - Protection : whether page can be read from, written to, executed from
    - Present : Indicates whether page is present in physical memory
    - Dirty : Whether page has been modified since brought into memory
    - Reference/Accessed : Whether page has been accessed (recently used!)
- Large pages can suffer from internal fragmentation, while small pages would require extremely large lookup tables for address translation.
- Multiple processes can share a PFN.
- So, the whole address translation process follows the following steps :
  - Identify VPN and offset (The bit length of offset is calculated using the size of the page, remaining first bits become VPN)
  - Index into the array of PTE as pointed by the page table base register (PTBR)
  - Get PTE from memory, extract PFN using VPN
  - Add offset to PFN to get to final address
- Address Space Identifier (ASID) : 8-bit field associated with each TLB entry (in TLB table) to distinguish mappings for various processes during context switch (one can flush with each switch also, but increased cache miss).
- **Translation Lookaside Buffer (TLB)** : Caches VA to PA mappings, saving a lot of costly memory accesses (plus parallel search by hardware).

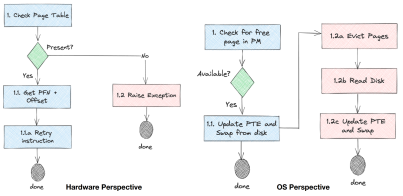


  - Goal : minimise cache misses. Locality matters :
    - Spatial : nearby addresses
    - Temporal : recently accessed addresses
  - For TLB misses, CISC hardware walks the entire page table in parallel to find the PTE, updates the TLB, and retries the instruction. In contrast, RISC software handles the trap by looking up the page table, updating the TLB via privileged instructions, and returning from the trap. Unlike normal traps, the hardware re-executes the faulting instruction after a TLB miss instead of moving to the next one.
- **Multi-Level Page Tables** : Tree-like page table structure



  - If an entire page is full of invalid entries, don't allocate that page of the page table at all. Use page directory (simple list) to track.
  - Easier to manage memory (each portion of the page table fits neatly within a page) allowing the OS to simply grab the next free page.
  - But, during a TLB miss, this approach requires two

loads from the memory (one for the directory and then for PTE itself).
    - Not just limited to two levels, can have deeper tree structure (with directory itself divided into multiple pages). Would require multiple memory access
- **Inverted Page Tables** : Instead of having one page table per process , have one single page table for all the processes. And, instead of VPN to PFN mapping, we have PFN to VPN (grows with physical memory size, instead of virtual AS)
    - Searching for an entry would require looking up the entire table (linear bad, efficient data structure needed)
- **Swap Space** : Dedicated space in memory which can be used to swap in and out pages. Allows OS to give perception that process has abundant memory.



    - Page Fault : Act of accessing a page that is not there in the physical memory (page table). Happens when the present bit is 0.
        - Causes the hardware raises an exception and the OS services using Page Fault Handler (piece of code), which searches through PTEs and gets the address from the PFN. Possibility of context switch.
    - Page Replacement : Process of swapping pages in/out from/of memory. Various policies to decide which page to evict (goal : maximise hits)
        - Optimal : Replace the page that will be accessed farthest in future (ideal, but not practical, to know the future).
        - First-In-First-Out (FIFO) : Evict the page that came first
        - Least Recently Used (LRU) : Replaces the least recently used page, works well due to temporal locality. Hardware support needed to know the LRU page, as software is not always involved. Use accessed or dirty bit (better)
        - Random : Random page gets evicted
    - If a page has been modified, it has to be pushed to disk. If not, the page is clean and can simply be replaced - less overhead!
        - Clock algorithm : Scans for unused and clean pages for replacement; and then moves to evicting unused and dirty pages.
    - Cold-start / compulsory miss : First few missed access, due to empty cache
    - Belady's Anomaly : Increasing cache size doesn't always mean improvement
    - Average Memory Access Time (AMAT) = (Hit% * T_M) + (Miss% * T_D)
    - Thrashing : When memory demands of processes exceed available physical memory, leading to constant paging. Might require killing some processes

➔ **Hybrid Approach** : Instead of having one page table per process, we have it for every segment (total 3 pages). Base register stores the start of the page table corresponding to the segment and bound indicates the end of the page table.
- First two bits are used for signifying segment, followed by VPN and offset.
- Unallocated size between stack and heap no longer takes up space in the page table, but variable page size now causes external fragmentation again.

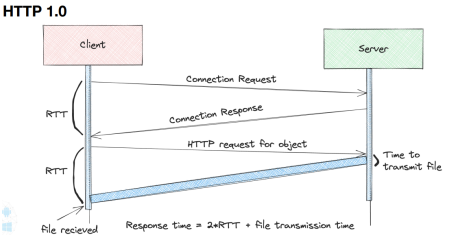
1.5 Network Application Architecture

➔ Two main types of network architecture :
- **Client-Server** : Clients request services from the server (an always-on host, with generally fixed IP/domain). Clients can always connect by sending packets to the server IP address. Often a single server is not enough.
- **Peer2Peer (P2P)** : No dedicated always-on system, peers communicate among each other (as client / server). Self-scalable and cost-effective, but unreliable, insecure and performance issues.

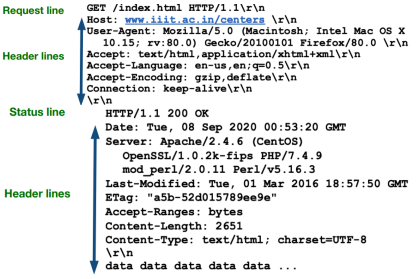
➔ Application layer protocols (eg: HTTP, SMTP, DNS, etc) define the types and syntax of message exchanges (request/response), semantics of the fields and when & how the process sends and responds to messages

➔ **Hyper Text Transfer Protocol (HTTP)** : Application layer protocol of the web, that defines the structure of the messages.

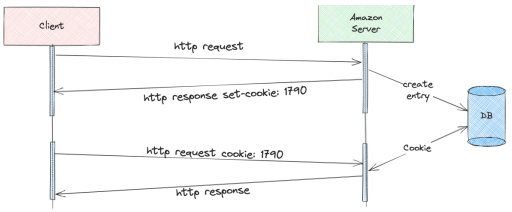
- Two types of HTTP connections :
  - Non-persistent (HTTP/1.0) : For every connection, the client has to create a request (one page may require multiple objects). This open connection closes after each request response.
  - Persistent (HTTP/1.1) : One connection for all the objects. The open connection is maintained.



- Two types of HTTP messages :
  - Request : Contains request line : Method (GET, POST, PUT, HEAD, DELETE), URL (server address) and Version (HTTP)
  - Response : Contains status line: Version and Status code (200: ok, 301: object moved, 400: bad request, 404: not found, 505: version not supported)

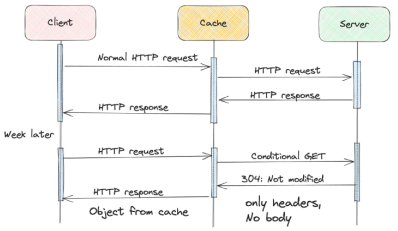


- HTTP server is stateless (every connection is treated separately), helps in supporting simultaneous connections.
  - But, websites may want to identify users (keep session information). HTTP header consists of information for **cookies**. Consists of four components : Cookie header line in HTTP response and request message, cookie file kept in client's system and backend database on the server/website.



- We also have HTTP 2.0 (standardised in 2015. Enable request response multiplexing over single TCP, request prioritisation , server push and HTTP header fields compression) and even 3.0 (underway).

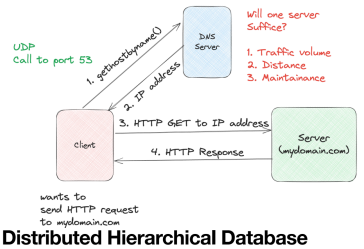
➔ **Web Caches** : Don't need to access the main web server every time. Can have proxy servers that satisfy requests on behalf of the main server. Browser can be controlled to point towards a cache (mentioned in response header).



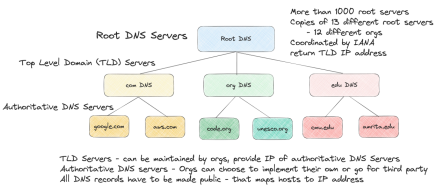
  - Reduces response time (often closer to client, reducing access link delay) and reduces traffic on main server (improving performance).
  - The cache copy might get stale. Conditional GET (HTTP request) used to verify if an object is up-to-date.

➔ **Content Distribution Networks (CDN)** : Global network of servers / data centers located around the world, to deliver web content to users quickly, reliably, and securely. CDNs adopt two different server placement strategies :
  - Enter Deep : Deploy server clusters in all access ISPs. High maintenance, higher throughput and lower delays
  - Bring Home : Building larger clusters at a smaller number of sites. Lower maintenance, lower throughput and higher delays.

➔ **Domain Name System (DNS)** : Directory service of the internet that translates hostnames to IP addresses.

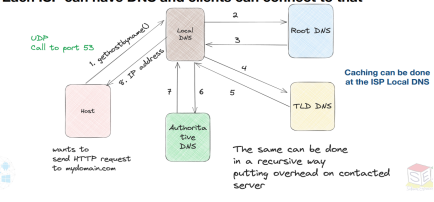


DNS: Distributed Hierarchical Database



Local DNS

Each ISP can have DNS and clients can connect to that



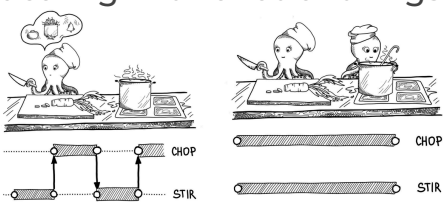
  - DNS servers are UNIX machines running Berkley Internet Name Domain (BIND) software. Runs over UDP (port 53) and provides the following services :
    - Host aliasing : A single host can have multiple aliases, resolve the names (get canonical names of host)
    - Mail Server Aliasing : Mail servers may also have aliases. Provide canonical names of mail servers
    - Load distribution : Perform distribution among replicated servers
  - DNS Servers store Resource Records (RR) and each RR is a tuple of form (name, value, type, ttl). Ttl (time-to-live) is how long the record can be cached

(in sec). Type can be one of the following :

- A : Name is hostname and value is IP address. (abc.com, 122.x.x.x, A, 3600)
- MX : name is domain and value is name of SMTP mail server. (x.com, mail.x.com, MX, 3600)
- CNAME : name is alias of canonical name and value is canonical name. (abc.com, x-abc.com, CNAME, 86400)
- NS : name is domain and value is hostname of authoritative DNS. (abc.com, ns.host.com, NS, 86400)

1.6 Concurrency

➔ Often confused with **parallelism**, concurrency is about dealing with a lot of things at once (interleaving process execution) while parallelism is doing a lot of things at once (different processes running in parallel across various CPU cores).



- Parallelism can be thought of as subclass of concurrency, but none implies the other

➔ **Threads** : Lightweight copy of the process that executes independently. Same process threads share the same code, variables, address space (and page tables). However, each thread has separate PC and function call stack management.

- Different from fork()-ed processes, as parent and child do not share any memory (essentially two different processes).
- OS schedules threads that are ready, similar to scheduling processes. Thread context (PC, registers) is saved into/restored from Thread Control Block (TCB).
  - Every PCB can have one or more linked TCBs corresponding to threads
- Kernel level processes have kernel level threads, which execute in kernel mode.

➔ **Race Condition** : When multiple threads executing concurrently and result depends on order of execution (non-deterministic in nature), interrupts and switch

- Critical Section : Code that is shared between the threads (leading to race conditions). Generally, shared variables / data.
- The solution is mutual exclusion, when one thread is accessing the critical section, others should wait. We need some synchronization primitives (hardware + software support) that ensure atomicity (similar to instruction level) and also that every thread gets access (avoid starving).

1.6.1 Locks

➔ Simple variable holding the state of lock at any instant of time, which can be Available (no threads hold the lock) or Acquired (Lock not available, one thread is holding it and in CS). Can hold further info like which thread holds the lock, create a queue for threads to get locks, etc.

- Owner : The thread that holds the lock. Owners need to free the lock for other threads to acquire it and access the critical section of the code.\
- Any lock should be mutually exclusive (prevent multiple threads from entering CS at same time), fair (each thread gets a fair chance to enter into the CS) and performance (not much overhead).
- Can't just simply disable interrupts inside a lock. Too much privilege to any random user program which can monopolise the processor. Won't even work on a multi-processor system. Plus, code that masks / unmasks interrupts is executed slowly (inefficient).
- Simple software locks (variables like locked=True), would also fail, as the conditions (if locked==False) can be checked for multiple threads, before updating the actual value (instruction-level atomicity only); causing multiple threads to gain access to CS.
- Two types of waiting when the process is waiting for a lock release : Spin-Wait (constant checking for the lock, consumes CPU cycles and resources) and Block (yield/sleep the process and try again later, CPU can work on something else)
- Multiple types of locks :

- **Test-And-Set** : Simplest hardware primitive (atomic exchange instruction) enabling testing of old values while setting the new value. Ensures only one thread can hold the value, and the other keeps spinning. No fairness guarantee; and while it works well on multi-CPU (esp. #threads = #CPU) machines, significant overhead in single-CPU machines (esp. If the lock-holding process gets interrupted)
- **Compare-And-Swap** : Another hardware primitive, which tests the address value with expected, before updating the memory location.
- **Load-Linked & Store-Conditional (LL/SC)** : Similar to typical load operation that fetches a value from memory and places it in a register. Stores conditional success if no intermittent store to address has taken place. In case of success, it updates ptr to value and returns 1 else returns 0.
- **Fetch-and-Add** : Atomically increment a value while returning the old value at a particular address. Used to build ticket lock, which uses a combination of ticket and turn variables, instead of a single flag variable.
- Locks ensure that threads can get access to CS, but threads might want to check for some conditions while executing.

Using TestAndSet Lock

```
typedef struct __lock_t
{
    int flag
} lock_t;


void init (lock_t *lock)
{
  lock -> flag = 0;
}

void lock (lock_t *lock)
{
  while (TestAndSet(&lock-> flag, 1)==1)
    // Keep spinning
}

void unlock (lock_t *lock)
{
  lock -> flag = 0;
}

int TestAndSet(int *ptr, int new)
{
  int old = *ptr;
  *ptr = new;
  return old;
}
```

LL/SC for building locks

```
void lock(lock_t *lock)
{
  while (1)
  {
    while (LoadLinked(&lock->flag)==1)
    ; //Keep spinning

    if (StoreConditional(&lock->flag,1)==1)
    {
     //store is successful, retrun else repeat all
     again
      return;
    }
  }
}

void unlock(lock_t *lock)
{
  lock->flag = 0;
}

int LoadLinked (int *ptr)
{
  return *ptr;
}

int StoreConditional(int *ptr, int value)
{
  if (no one has updated ptr since load linked)
  {
    *ptr = value;
     return 1;
  }
  else
  {
    return 0;
  }
}
```

Compare-And-Swap

```
int CompareAndSwap(int *ptr, int expected, int new)
{
  int actual = *ptr;
  if (actual == expected)
  {
    *ptr = new;
  }
  return actual;
}
```

Inside the lock function the call will be:

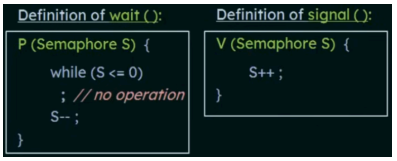
```
while(CompareAndSwap(&lock->flag, 0,1) ==1)
```

Ticket Lock

```
typedef struct __lock_t
{
    int ticket;
    int turn;
} lock_t;

void lock_init (lock_t *lock)
{
  lock_t -> ticket = 0;
  lock_t -> turn = 0;
}

int FetchAndAdd(int *ptr)
{
  int old = *ptr;
  *ptr = old + 1;
  return old;    //returns old value
}

void lock (lock_t *lock)
{
  int myturn = FetchAndAdd(&lock->ticket);
  //When ticket value = my turn, thread goes into CS
  while (lock->turn != myturn)
  {
    // keep spinning
  }
}

void unlock (lock_t *lock)
{
  // next waiting thread can enter CS
  FetchAndAdd(&lock->turn);
}
```

➔ **Condition Variables** : Explicit queues that the threads can put themselves on when a state of condition is not as desired. Eg: lock is not available. When the condition is met, thread can be woken up to continue.
- Allows signalling (passing info on condition) between threads
- Defined as pthread_cond_t c, where c is a condition variable with two ops :
  - wait() : when thread wants to put itself to sleep (due to some condn)
  - signal() : due to some change (in condn), the thread wants to wake up

➔ **Semaphores** : Single structure which can act as both lock and condition variable.
- A simple value shared between threads. Starting with a value equal to the number of resource instances available, it has two (atomic) routines :
  - sem_wait() : Wait while value is -ve. Then, decrement one and get access
  - sem_post() : After access done, increase the semaphore value
- Two types of semaphore :
  - Counting : Initial value set to as many resource instances available, allowing as many threads to get access to the CS at a time.
  - Binary : Counting semaphore with one resource instance (initial value is 1).
- Value of semaphore, when negative, equals to number of waiting threads



1.6.2 Concurrency Problems

➔ Note : Interchangeable use of threads and process. Hold for both.

➔ **Race Condition** : When multiple threads access shared resources concurrently, leading to unpredictable and undesirable outcomes due to the order of execution.
- Can be resolved using simple locks around the critical section

- **Deadlock** : Situation where two or more processes are blocked indefinitely, each waiting for a resource that the other holds. Creates circular dependency, preventing any of the processes from making progress.
  - Soln : Ensure no lock is held by a thread when it is waiting for some other lock.
  - Often preferable to avoid rather than prevent. Concepts like Scheduling (If OS knows which threads require locks at which point of time, it can schedule them accordingly) and Bankers algorithm (practically not applicable).
  - Many systems also employ deadlock detection (periodic cycle detections or if OS freezes) and recovery (reboot the system) techniques
- **Producer-Consumer / Bounded-Buffer Problem** : Using only one lock for managing various threads handling adding/taking to/from some buffer.
  - Causes issues like race conditions, wrong thread being signalled, everyone going to sleep, starvation, etc (due to spurious wakeup and ill-implementation).
  - Must use two locks, one for producer threads and other for consumer threads. Both lock access completion signals the other lock.
- **Readers-Writers Problem** : Multiple readers can access some resource, but only one writer at a time. Writers would starve.
  - Must add some sort of priority mechanism (queue), common to both.
- **Dining Philosophers** : There are N philosophers sitting around a table with a fork between each. They think for some time and then try to eat, by acquiring the two forks on either side. Represents threads trying to acquire shared resources.
  - Might lead to deadlocks (when, for instance, all philosophers acquire the left fork and then try to get the right one)
- Concurrency bugs can be broadly classified into two categories :
  - Deadlock bugs : Threads keep waiting for each other. 4 conditions that should together hold for a deadlock to occur :
    - Mutual Exclusion : Thread claims exclusive control of a resource (eg: lock)
      - Hardware primitives like Compare-and-swap (still chance of livelock)
    - Hold-and-wait : Thread holds a resource and is waiting for another
      - Can be avoided by using a master lock to hold all locks at once. Would impact performance and concurrency gains
    - No Preemption: Thread cannot be made to give up its resource (eg: the lock)
      - Try locks before actually getting them. Possibility of livelock, if other threads also follow the same order (can be resolved by adding delay)
    - Circular Wait: There exists a cycle in the resource dependency graph
      - Can be avoided by acquiring locks in a particular order
  - Non-deadlock bugs : Incorrect results when threads execute. Mostly of type :
    - Atomicity Violation : Critical section access by multiple threads
    - Order Violation : Assuming another thread has already run

1.7 Networking (contd.)

1.7.1 Link Layer

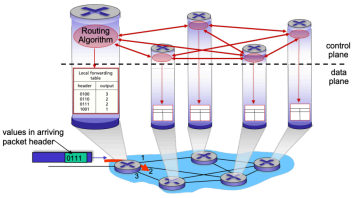
- **Subnets** : Dividing a network into one or more (hierarchical) networks
  - A subnet mask of 255.255.255.0 (or /24 in Classless Inter Domain Routing (CIDR) Notation, which is the number of 1s in the address) implies a network containing 254 host addresses (only the last part can change).
- The sender knows the receiver's IP, given the domain (through DNS), allowing L3 communication. For L2, we need a MAC address, though.
  - **Address Resolution Protocol (ARP)** : A table/cache, with each IP node (router, host), containing IP/MAC address mappings for some LAN nodes in the form <ip address, MAC address, TTL> (generally, ttl ≈ 20mins).
    - In case of same-network comms, the sender sends out an ARP query / request (broadcasted to all network nodes), which includes the sender's IP and MAC address, target IP and MAC address (set as FF:FF:FF:FF:FF:FF)
    - All the nodes store this (sender's) broadcast mapping and the target host

replies with its MAC (not a broadcast) for the sender to store.
- When communicating across networks, default gateway (router connecting to the outside of the network) IP (no broadcasting) is used (IP of 172.18.12.92).

1.7.2 Network Layer

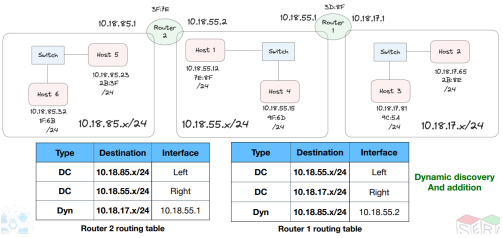
➔ Use IP address (logical address for unique identification within a network to forward packets to the intended destinations.
- Needs to identify the best path, a dynamic process that changes based on network conditions. Two sub-methods :
  - **Forwarding** : Move packets from router's input link to output link (next step)
  - **Routing** : Determine route from source to destination (full route)

➔ Network layer functions can be divided into two planes :



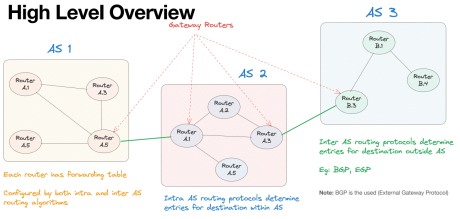
- Data plane : Local per-router function, determines how datagram arriving on router input port is forwarded to router output port
- Control plane : Network wide logic, determines how data is routed along end to end path from source to destination. Two approaches : Traditional routing algorithms and Software defined networking (SDN)

➔ Routers forward packets not explicitly addressed to them. They maintain a map of all networks they know about (given the destination IP), called **Routing Table**.



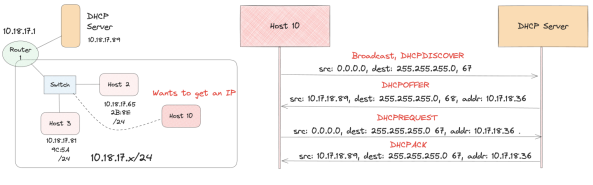
- While ARP tables are populated on the fly, routing tables need to be ready apriori (routers may drop packets if IP is not known). Three methods for population :
  - Directly Connected : Networks to which the router is directly attached to
  - Static Routes : Routes manually provided by an administrator
  - Dynamic routes : Routes automatically learned from other routers (various protocols like OSPF, BGP, EIGRP, IS-IS are used by routers to inform about the different networks they are connected to)
- But there are billions of destinations, not everything can be stored in each router. Sending so many links with each other can itself bring down the network. There are two parts to it: Internet (network of networks) and that each network admin may want to control routing in its own network

➔ **Autonomous Systems (AS)** : Regions of aggregate routers (aka domains)



- Total of around 70,000 AS's have been assigned (not all are active)
- We need mechanisms for handling routing within (Intra) and across (Inter) AS
- All routers in AS must run the same intra-domain protocol. There is a gateway outer at the edge of each AS which connects with the router in another AS.
  - Gateways perform inter-domain as well as intra-domain within their network
  - Intra-AS routing protocols : OSPF (Open Shortest Path First) Protocol (classic dijkstra-based link state routing), RIP (Routing Information Protocol), EIGRP
  - Inter-AS routing protocols : BGP (Broader Gateway protocol) based on path vector protocol (considered as "glue that holds internet together")

➔ How does the host get an IP address? Hard-coded by sysadmin in the config file (e.g., /etc/rc.config in UNIX) or dynamically generated by **DHCP (Dynamic Host Configuration Protocol)** when joining from a server.
- DHCP runs over UDP. Client uses port 68 and server port listens on port 67.
- DHCPDISCOVER is broadcasted to all nodes, including the DHCP servers.
- Multiple servers offer IP addresses, the client chooses one (first response) and broadcasts the acceptance.
- DHCP server can also give details like



address of DNS server, address of first hop router, network mask, etc.

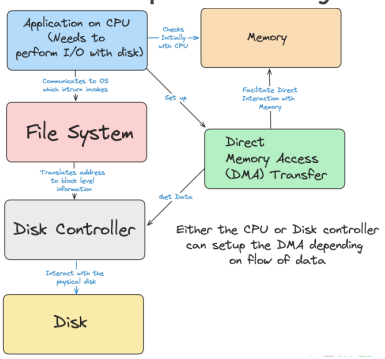
- ISPs get IP address blocks from ICANN(Internet Corporation for Assigned Names and Numbers) and allocate IP addresses through 5 regional registries (RRs).
  - There are not enough IPV4 addresses (Last chunk was allocated in 2011)
  - IPV6, the next update, comprises of 128 bit address space
  - IPV4 works for now due to **NAT (Network Address Translation)**
    - All devices in the network share just one IPV4 address (public IP) as far as the outside world is concerned.
    - When devices from a network want to communicate with an outside network, NAT modifies the source IP to its own public IP (to make it appear that communication is from the larger public IP) using a translation table (known as NAT or xlate-table).
    - Various types (each offering a distinct translation method) : Static NAT, Dynamic NAT, Port Address Translation or NAT Overload

1.8 Data Persistence

➔ Hardware (I/O device) and software (file system) needed to store data persistently

➔ The flow of access is as follows :

- Application performs read or write to a file
- CPU communicates to the OS which invokes the File System (FS). OS may check its cache if it's already there.
- FS prepares block level information to the disk controller. A Direct Memory Access (DMA) is set up.
- Disk controller performs the physical read or write based on commands from DMA and file system
- Read : Disk → DMA ; Write : DMA → Disk



1.8.1 Hard Disk

➔ Consists of 512 byte-sized blocks, each of which can be read or written

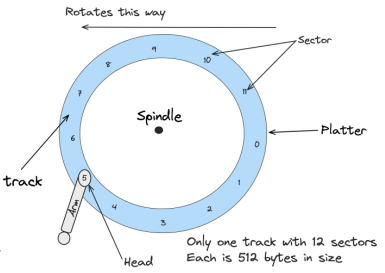
➔ Sectors are numbered from 0 to n-1 on disk with n sectors - Address space

➔ While many file systems read / write 4 KB at a time (or more), the singular 512-byte block writes are guaranteed atomic (happens completely or not at all).

➔ Torn write : power loss during the operation resulting in incomplete write.

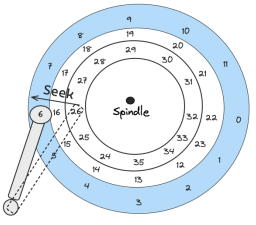
➔ As for the geometry of the disk :

- Platter is the circular hard surface, which has 2 sides
- Each surface has thousands of tightly packed tracks in the shape of concentric circles.
- Each track consists of multiple sectors, spread around. These sectors contain the encoded data.
- The arm, with the head at its end, moves across to allow read / write. Spindle connected to a motor, spins the platter around at a fixed constant rate (7200 to 15000 RPM → ~6ms full rotation time)



➔ Three key phases in the process of disk access :

- **Rotation** : The head can read only when the desired block is under it, and has to wait for the disk to rotate. This is known as rotational delay.
- **Seek** : The disk contains multiple tracks and the head must move across tracks. This is a costly operation as it consists of multiple phases : Acceleration, Coasting and Settling (settling alone can take upto 2ms)
- **Transfer** : Finally, read / write from the surface.



➔ Some metrics are defined as :

- Total Time : $T_{I/O} = T_{Seek} + T_{Rotation} + T_{Transfer}$
- Rate : $R_{I/O} = Size_{Transfer} / T_{I/O}$

➔ By the time the head moves (from one track to another), the desired block in the track would have rotated. To avoid this the beginning of the

Some Analysis

Workload	Metric	Cheetah	Barracuda
Random (4 KB reads)	Tseek	4 ms	9 ms
	Trotation	2 ms	4.2 ms
	Ttransfer	30 micros	38 micros
	TI/O	6 ms	13.2 ms
	RI/O	0.66 MB/s	0.31 MB/s
Sequential (100 MB reads)	Ttransfer	800 ms	950 ms
	TI/O	806 ms	936.2 ms
	RI/O	125 MB/s	105 MB/s

There is large difference in performance between **high-end performance** drives and **low-end capacity** drives

next track is slightly offset or **skewed**.

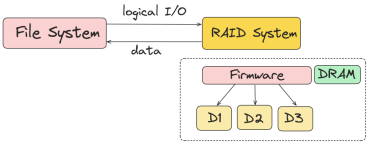
➔ Modern disk drives also have cache (~8-16 MB), often referred to as **track buffer**. It allows the driver to quickly respond to requests.
- When reading from a sector, cache all sectors in that track for faster subsequent reads. For writes, there are two choices :
  - Writeback (Immediate Reporting) : Acknowledges the write as completed as soon as the data reaches cache memory. Gives the illusion of very fast driver, but might cause issues, especially if order needs to be preserved
  - Writethrough : Acknowledge after the write has been written to the disk. Data is written to cache and disk together, causing performance issues.

➔ Coming to **workload distribution**, we have two types :
- Random Workload : Issues small (4 KB) reads to random locations on the disk. Common in applications like DBMS.
- Sequential Workload : Reads a large number of sectors consecutively from disk. Common in applications like backups, streaming, etc.

➔ **Disk scheduler** decides which request to schedule next to improve performance, using the time estimates (based on delays) for each request.
- Shortest Seek Time First (SSTF) : Orders the queue of I/O requests by the nearest track to the current position. Might cause starvation in case of a steady stream from two adjacent tracks. Also requires geometry knowledge, which is not known to the OS (One solution : Implement something like Nearest Block First).
- Elevator / SCAN : Simply move back and forth, servicing the requests in order. If a request comes for a block on a track that has already been serviced in this sweep (pass across the disk), it has to wait in a queue till the next sweep. Two categories :
  - F-SCAN : Freeze the queue to be serviced when doing a sweep. Avoids starvation of far-away requests.
  - C-SCAN (Circular scan) : Sweep from inner-to-outer and outer-to-inner, etc
- Shortest Positioning Time First : Considers rotation and seek delays to prioritise faster starts (short position time). Depends if seek is faster / slower than rotation

1.8.2 Redundant Arrays of Inexpensive/Independent Disks

➔ Simply known as **RAID**, it consists of techniques to use multiple disks in concert to build faster (parallel ops), bigger (larger memory capacity) and more reliable (redundancy and backup options) disk systems.

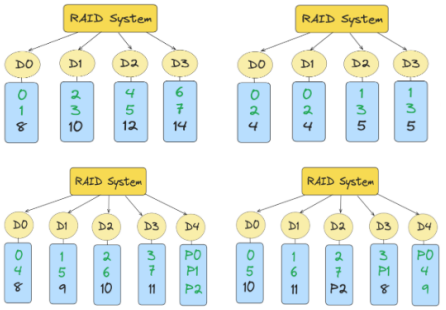


➔ Externally (to the OS) it is like any normal group of memory blocks one can read/write to/from; but internally it is a very complex structure consisting of multiple disks, own memory (volatile and non-volatile for various buffer use and parity calculations) and processors/microcontrollers to manage the whole system.

➔ Any raid approach is evaluated based on the Capacity, Reliability (how many failures/faults can the RAID system tolerate) and Performance (impact of different workload on the latency, throughput and rate of I/O). Two main things to evaluate :
- Single-request Latency : Latency of single I/O request to RAID
- Steady-state Throughput : Total bandwidth of concurrent requests

➔ We have the following main RAID levels (assume disk transfers at S MB/s under sequential and R MB/s under random; and we have N B-block sized disks) :
- **Striping** (RAID Level 0) : Spread the blocks across the disks in a round robin fashion, without any redundancy. The number of consecutive blocks in a disk are called chunks; and chunks in the same row together constitute a stripe.
  - Too small of a chunk size will cause strips content across disks, increasing parallel reads/writes (increasing the positioning delay also) and too big reduces parallelism causing multiple concurrent requests to achieve high throughput. No best size.
  - Steady-state throughput equals N*S MB/s for sequential and N*R MB/s for random workload. It is more like an upper bound

- **Mirroring** (RAID Level 1) : Each entry has a copy placed in a different disk, enabling it to handle single failures for a copy. Data is stripped across mirrored pairs. The system can use any of the copies for read, but write has to happen to both copies (may happen in parallel, similar time but higher due to the worst case of the two). Capacity also becomes N/2 due to the copies.
  - Each disk will only deliver half the peak bandwidth, (N/2)*S for sequential reads (as the two disks divide the workload). And since each sequential write requires writing in two different locations, the bandwidth during sequential write is (N/2)*S (again half the peak). Random reads : N*R; and write : (N/2)*R.
- **Parity** (RAID Level 4) : For each stripe of data, a parity (generally XOR) block is added. Allows for detection / handling of single failure across the system. Just one disk is used for parity, leaving the whole (N-1)*B memory for actual data.
  - During write, especially random write, we also need to update the parity. Naive approach would be to iterate the whole stripe and calculate new parity (additive approach). But, it's rather better to just flip the parity bits if the updation bits are different from before (subtractive approach).
  - Random reads, sequential reads and writes have a bandwidth of (N-1)*S, assuming parity writes happen in parallel. But for random writes, it becomes R/2 due to the parity disk becoming a bottleneck.
- **Rotating Parity** (RAID Level 5) : Similar to RAID4, but removes the parity bottleneck, by spreading the parity bits across numerous disks.
  - Similar sequential read and write performance as RAID4: (N-1)*S. Random reads slightly better as it now utilises all disks. As for random write, we can now parallelised due to distributed parities , providing a max bandwidth of (N/4)*R MB/s (4 due to the 2 reads + 2writes).



➔ To summarise :
- RAID0 : Performance, but no reliability
- RAID1 : Random I/O performance and reliability
- RAID4 : Performance and steady-throughput, but terrible random writes
- RAID5 : Capacity, reliability and sequential I/O

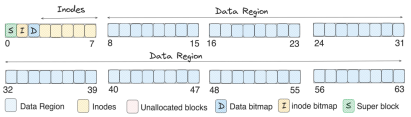
1.8.3 Storage Virtualisation

➔ **Files** : Linear array of bytes each of which can be read or written
- Each file has a unique low-level name (OS given), called inode number (i-number) apart from the human-given name.
- OS is not concerned about the file types (image, code, etc); applications can care
- Everything in Unix is virtually a file

➔ **Directory** : A special kind of file (has its own inode number) containing mapping of file names and their inode numbers for the various files and directories it contains.
- Each directory has two extra files : "." for current dir and ".." for parent dir.
- Unix Directory Tree : Files and directories arranged in a tree structure, with the root directory referred to as "/". Uses a separator (generally "/") to name subsequent directories. Everything is an abstraction by OS

➔ **File System** : Organization of files and directories on disk. Pure software
- It should ensure that data is stored persistently, and retrieved when requested. Responsibility of making sense out of the 0s and 1s stored in memory.
- It mainly has to provide three interfaces :
  - Creation of files : Support creating files, allocate space
  - Accessing files : Reading and writing files
  - Deletion of files : Delete files and clear space
- It also stores some metadata about the files in a structure called inode (referenced by the inode number). It includes data such as file type, file size, last access, last modified, protection information, etc.
  - Each inode needs to track disk blocks (not necessarily contiguous) of the file.
  - Stores pointers to the disk block, called direct pointers. In case of large file

size (cant store all the direct pointers), use an indirect pointer which points to a block containing more pointers (called indirect data block). The indirect block is allocated from the data region.
    - Each indirect pointer can further point to an indirect data block.
- An OS can have more than one file system
- Lets try building a **Very Simple File System (VSFS)** :



    - Data Structure : Some blocks (exposed by disk) need to be reserved for the metadata and rest for the actual data.
        - Apart from the inode table in the metadata, we also store bitmaps to denote if the corresponding block is free or not.
        - Lastly, we have a super block, holding the entire organisation (which blocks are what, type of file system, etc). During the mount, OS reads this super block to initialise various parameters.
- But how does a FS manage access to files?
    - Kernel uses a set of data structures to track all open files :
        - Global open file table : One entry for every open file (also stores sockets, pipes, etc.), pointing to the in-memory inode of the file.
        - Per-process open file table : Array of all the files that the process has opened, indexed using the file descriptor (FD). Every process has three files (stdin, stdout, err); with FD 0, 1 and 2 respectively; open by default.
        - Per process file entry -> global file table entry -> inode of file.
    - For opening a file :
        - Traverse the path name by locating desired inode, using the i-number and recursively iterating the dirs from the root to the desired file / dir.
        - Finally, returns a file descriptor (FD) which points to in-memory inode. The FD also acts like an offset value, allowing random access to the file.
        - In the case of a new file, new inode and data blocks will be allocated using bitmap and update directory entry.
        - Open system call creates entries in both tables and returns the FD.
    - For reading a file :
        - Read in the first data block of the file with help of inode. Update the last accessed time in the inode and in-memory open file table for file offset , file descriptor, etc. Repeat the process for reading each block of data
        - Just deallocate (no disk I/O) the FD, once the file is closed.
    - For writing to a file :
        - Total of five I/O : Read and write to data bitmap, read and write the inode and finally write to the actual block itself.
        - Even more in the case of creation of a new file.

➔ **Free Space Management** : Apart from bitmap for tracking free blocks, we can have pointers in the super block pointing to the first free block which can then point to the next free block and so on. This is referred to as the free list.
- Sequence of data blocks are allocated contiguously for performance. Pre-allocation policy is commonly used heuristic when allocating data blocks

➔ **Caching / Buffering** : Reading / Writing are expensive ops. (deeply located files can get upto 100 I/Os). To minimise such effects, the system memory can be used to cache important blocks (minimise overhead, also).
- Modern systems employ dynamic partitioning of memory by integrating virtual memory pages and FS pages into unified page cache.
- Writes are a little tricky as at some point the disk has to be accessed to store the data. Buffering (adding delay to perform batch I/O) and then scheduling the I/Os in a particular order for performance gain.
- At the end its all about trade-off's : Durability (DBs) vs Performance (PCs)

1.9 Miscellaneous

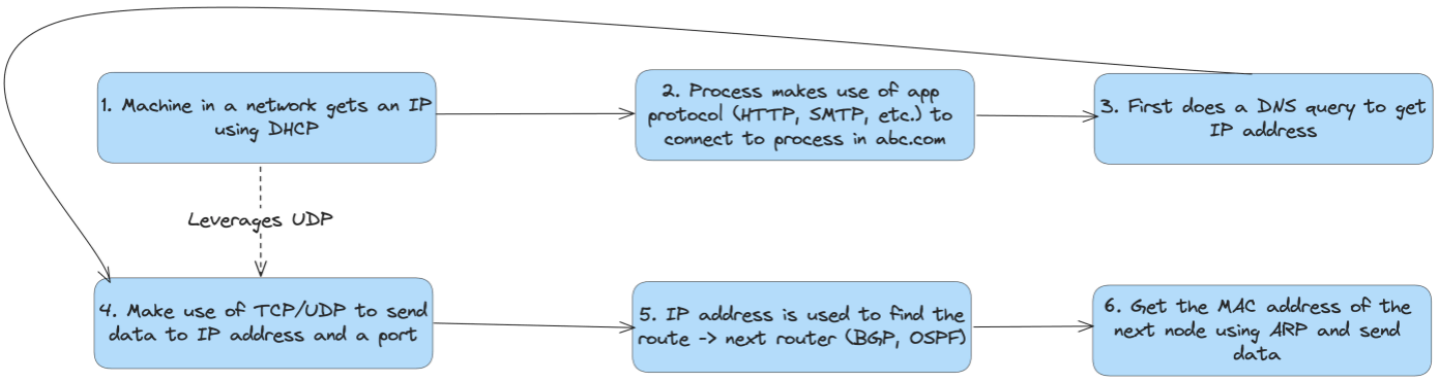
- In a nutshell : https://iiithydresearch-my.sharepoint.com/?login_hint=aviral%2Egupta%40research%2Eiiit%2Eac%2Ein&id=%2Fpersonal%2Faviral%5Fgupta%5Fresearch%5Fiiit%5Fac%5Fin%2FDocuments%2FAcads%2FUG2%2FSem3%2FOSN%2FLectures%2FOSN%5FL26%2Epdf&parentview=1
- Some good resources (to understand what I cannot with mere text) :
  - LL/SC lock : Load Linked Store Conditional - Georgia Tech - HPCA: Part 5
  - Semaphores : Semaphores
  - Concurrency Problems : Associated slides must (well-explained with example)
- A ‘good’ analogy to spin-waiting vs blocking, when waiting for a lock can be a single restroom being shared by many people at an office. When it is occupied, one might stand at the gate, constantly knocking until it is vacant (wasting time, not being productive) or one can go back and check back in later after some time (no time waste, but allow some other person, who comes later, to get access before).
- Some other terms used in the text, that might benefit with more details :
  - **Livelock** : Concurrency issue where multiple processes/threads repeatedly change their state in response to each other, but none of them make any forward progress.
    - Unlike deadlock, where processes are blocked, in livelock, the processes are active but their actions continuously negate each other, preventing them from completing their tasks.
  - **Banker’s Algorithm** : A resource allocation and deadlock avoidance algorithm, ensuring that a system remains in a "safe state" by simulating the allocation of resources and checking if all processes can complete their execution without entering a deadlock
  - **Clock Algorithm** : Widely used page replacement algorithm that aims to mimic the behavior of the LRU algorithm without much overhead associated with tracking the least recently used page. Here's how it works :
    - Pages in memory are organized in a circular list, like a clock face, with a reference bit for each page initially set to 0 (when loaded). When a page in memory is accessed, its R-bit is set to 1.
    - When a page fault occurs and a page needs to be replaced, the clock hand (a pointer moving around the circular list), points to a page and checks its R-bit.
      - If R-bit is 0, the page has not been accessed recently and is replaced with the new page’s R-bit set to 1. The hand then moves to the next page.
      - If R-bit is 1, the page has been accessed recently. Its R-bit is reset to 0 and it gets a "second chance". And the hand advances to the next page.
    - The advanced version makes use of the dirty bit, instead.
- Some other random content :
  - On a LAN, each network interface (like a network card in a computer) has a globally unique MAC address and a locally unique IP address.
  - MAC Address : 12 hex digits of form “00:1A:2B:3C:4D:5E. The first three identify manufacturer (IEEE) and the next three are uniquely assigned by manufacturer.
  - Two types of file references in a file system :
    - Hard Links : Create another file that points to the same inode. Essentially both files have the same underlying data, just different user-given names.
      - User can only unlink files, the OS decides when to delete (when no more files are linked to it).
      - Limited as directory links are not possible, can't link to files in other disks as inode is unique within a file system only).
    - Soft / Symbolic Links : Create a file by itself with a different inode number. If the main file is deleted, link points to an invalid entry: dangling reference

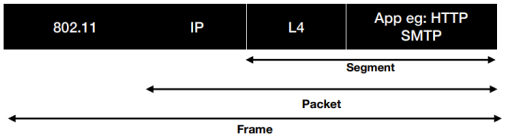
- Servers are often bind to “well-known-ports”, while clients are assigned ephemeral ports by OS temporarily. Some common ports for various protocols :
  - FTP (file transfer) : 20, 21
  - SSH (remote login) : 22
  - SMTP (email) : 25
  - HTTP (world wide web) : 80
  - HTTPS (secured web) : 443
  - RTSP (media player control) : 543
- The Internet is just a bunch of routers. Internet vs internet :
  - internet : distinct interconnected networks (network of networks)
  - Internet : refers to the specific, globally recognized network we all use
- Earlier, network broadcasting used to happen on a single line (only one host could transmit successfully at a time). Status allocation techniques were used (every machine gets some time to transmit / receive) using round robin scheduling. Packet collision threats (try again after random delay).
- Mounting a filesystem refers to making files and directories on a storage device, like a hard drive, accessible to the operating system and users. A device can have several storage devices with their separate file systems.
- Normal writes put the data into a buffer, and at some point it will be written to persistent storage (using system calls like fsync). This is done for performance enhancement.
  - Reading and writing happens sequentially by default. Successive read/write calls fetch from the offset that is being used. Calls like lseek for random accesses (by manipulating the offset).
- For TLB misses, CISC hardware walks the entire page table in parallel to find the PTE, updates the TLB, and retries the instruction. In contrast, RISC software handles the trap by looking up the page table, updating the TLB via privileged instructions, and returning from the trap. Unlike normal traps, the hardware re-executes the faulting instruction after a TLB miss instead of moving to the next one.
- From a developer perspective, network architecture is fixed, while application architecture is something that can be controlled.
- Transport layer provides logical communication between application processes running on different hosts, with the help of protocol actions in the end systems (sender breaks application messages into segments before passing to network layer and receiver reassembles messages before passing to application layer)
- Every process assumes access to a block of memory from 0 to MAX (all in VAS).
- Hosts are any nodes that are not a router. They can discard packets due to reasons like intended discard, errors, firewall rules, overflow/congestion, etc
- Coarse-grained vs Fine-grained memory management :
  - Coarse-grained : Fewer large-sized segments. Lower overhead
  - Fine-grained : More of smaller sized segments. Precise customisation

➔ Some calculation examples :

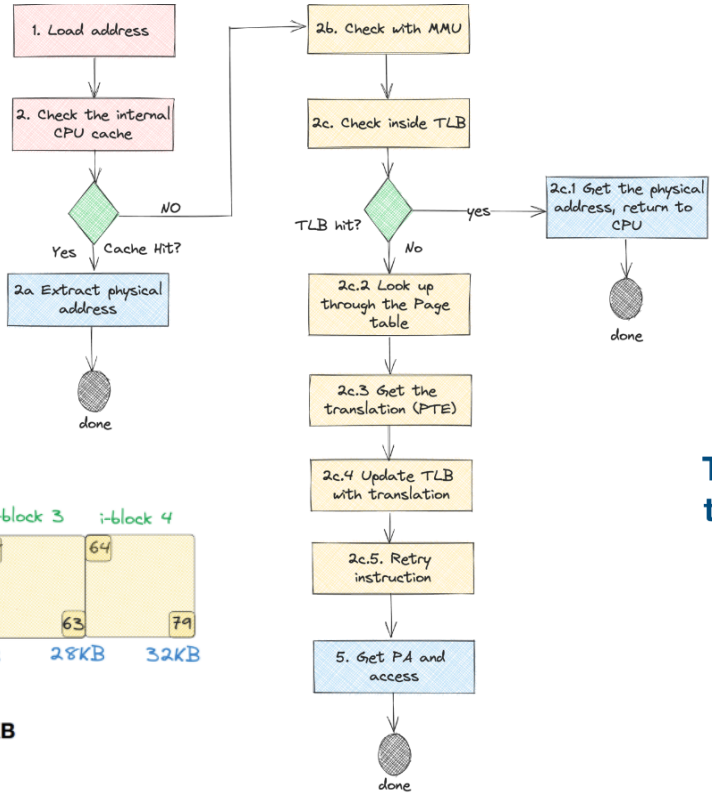
- Given a 32-bit address space with 4KB pages, find the size of page table size assuming 100 processes :
  - 4KB (2^{12}) page => 12 bits, leaving 20 bits for VPN (2^{20} mappings possible)
  - Assuming 4bytes for each mapping => $4x2^{20}$ = 4MB per process per page
  - For 100 processes, this gives us 400MB

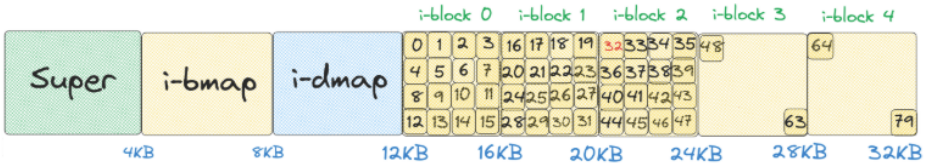
1.X Glossary





- Segments carry data across the network
- **Segments** are carried within the packets, within frames
- Each layer adds a **header** (Above L4 will be replaced by its header)





- Using inode number, FS can locate inode, eg: inode number: 32
- Calculate offset into inode: 32 X (sizeof(inode)) = 32 * 256 = **8192 => 8 KB**
- Add offset with start address of inode = 12KB + 8KB = **20KB**

Writing a File To Disk

Reading a File From Disk