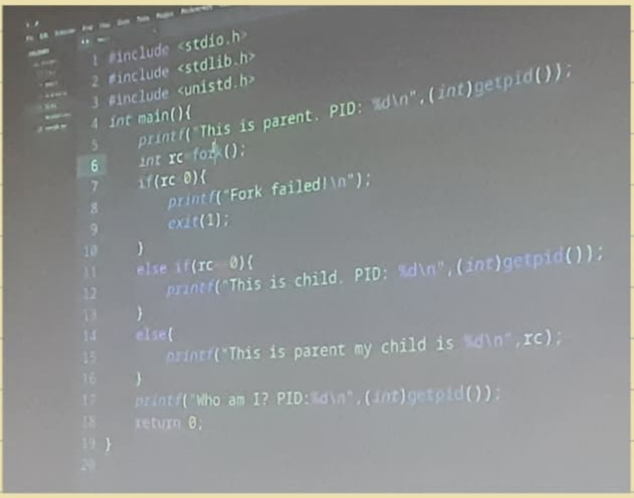
1/8

OSN

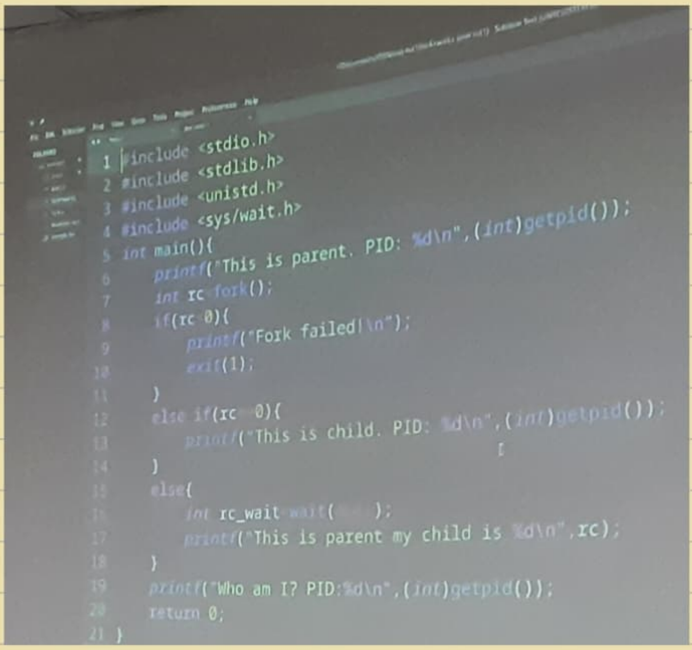
OS : Coordinates everything
↳ Middleman b/w Hardware and user

OS handles interactions with the disk and performs storage management.

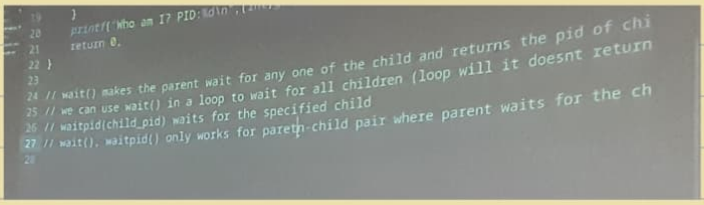
2/8



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(){
    printf("This is parent. PID: %d\n",(int)getpid());
    int rc=fork();
    if(rc<0){
        printf("Fork failed!\n");
        exit(1);
    }
    else if(rc==0){
        printf("This is child. PID: %d\n",(int)getpid());
    }
    else{
        printf("This is parent my child is %d\n",rc);
    }
    printf("Who am I? PID:%d\n",(int)getpid());
    return 0;
}
```



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(){
    printf("This is parent. PID: %d\n",(int)getpid());
    int rc=fork();
    if(rc<0){
        printf("Fork failed!\n");
        exit(1);
    }
    else if(rc==0){
        printf("This is child. PID: %d\n",(int)getpid());
    }
    else{
        int rc_wait=wait(NULL);
        printf("This is parent my child is %d\n",rc);
    }
    printf("Who am I? PID:%d\n",(int)getpid());
    return 0;
}
```



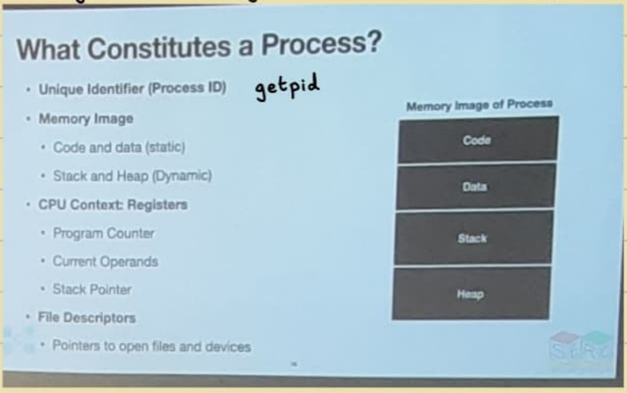
```c
    }
    printf("Who am I? PID:%d\n",(int
    return 0;
}

// wait() makes the parent wait for any one of the child and returns the pid of chi
// we can use wait() in a loop to wait for all children (loop will it doesnt return
// waitpid(child_pid) waits for the specified child
// wait(), waitpid() only works for pareth-child pair where parent waits for the ch
```

5/8

OS → divides CPU work, makes it feel like any process has infinite access to CPU.
Process → Takes resources

- Memory :
  - Static Memory → Variables [Stack]
  - Dynamic Memory → malloc, calloc [Heap]



What Constitutes a Process?

- Unique Identifier (Process ID) getpid
- Memory Image
  - Code and data (static)
  - Stack and Heap (Dynamic)
- CPU Context: Registers
  - Program Counter
  - Current Operands
  - Stack Pointer
- File Descriptors
  - Pointers to open files and devices

Memory Image of Process
Code
Data
Stack
Heap

Create memory image for the process
⇓
Allocate PC, SP

- Kernel : Handles direct instructions with hardware
  ↪ Part of the code

$$\begin{tikzcd}
 & \text{Kernel} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Monolithic} & \text{Hybrid} & \text{Micro}
\end{tikzcd}$$

Monolithic (If one fails, all fail)

Hybrid [Abstracted]

Micro (Only file access is abstracted)

$\underbrace{\text{Kernel}}$ has stack
↪ Process

- API ⟶ $\underbrace{\text{System calls}}$ (fork, exec, wait)
  ↪ Resides in Kernel

- Limited Direct Execution (LDE) ⟶ CPU

fork ( ) ⟶ System call
↓
Kernel

} IDT ↖

2	0x100
3	0x200
4	

$\underbrace{\text{System call}}$ ⇒ Mode change ⇒ User mode to Kernel Mode
↪ TRAP Instruction : Changes privilege ↗

$\underbrace{\text{IDT}}$ ⇒ OS code gets executed
↪ Interrupt disrupt Table

Context ⟶ PC, SP ⟶ Saved

User stack ✗
$\underbrace{\text{Kernel stack}}$ ✓
↪ Per process level

10
11 syscall ⟵ OS
12

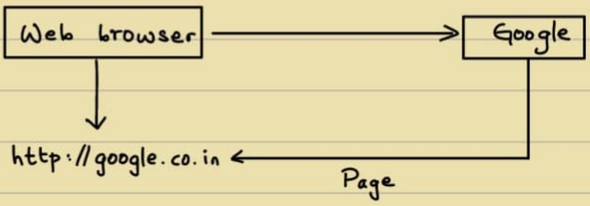
$$\begin{tikzcd}
 & \text{Interrupt} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\text{I/O Interrupt} & \text{Hardware Interrupt} & \text{Network Interrupt}
\end{tikzcd}$$

- <u>Context</u> <u>Switch</u>:

  How to decide which process to run next ⟶ Scheduling

  In reality, Jobs can arrive at any time.

<u>26/8</u>



Port → 80

Process id	Port

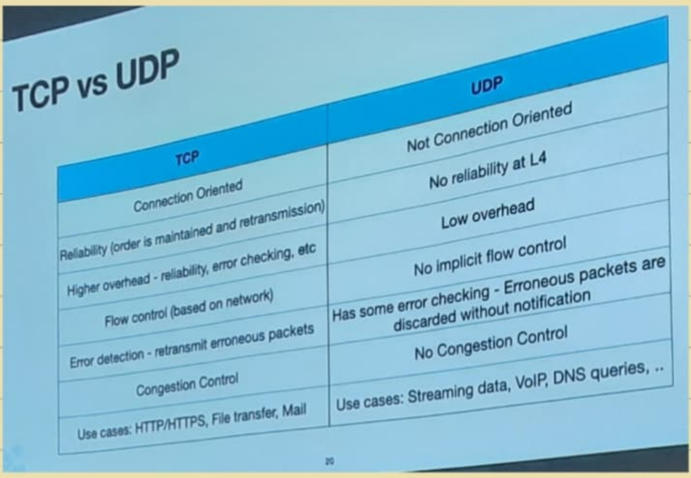
← Mapping

- Socket ⟶ Abstraction ⟶ Allows communication
  ↳ API

Stream → continuous (TCP)
↳ Mutual connection (strict) + Connection oriented

Datagram ⟶ No memory + No persistent connection (Stateless connection)
↳ UDP
↳ Unrealiable (by default)

☆



TCP vs UDP

TCP	UDP
Connection Oriented	Not Connection Oriented
Reliability (order is maintained and retransmission)	No reliability at L4
Higher overhead - reliability, error checking, etc	Low overhead
Flow control (based on network)	No implicit flow control
Error detection - retransmit erroneous packets	Has some error checking - Erroneous packets are discarded without notification
Congestion Control	No Congestion Control
Use cases: HTTP/HTTPS, File transfer, Mail	Use cases: Streaming data, VoIP, DNS queries, ..

20

Flow control : Control speed of data transmission.

2/9

- UDP $\longrightarrow$ Unreliable at L4 $\because$ Fire and Forget, i.e. responsible for giving data but not ensuring recieving.

Headers differentiate TCP and UDP.
↳ 32 bits ↳ lightweight (8 bytes)

TCP $\longrightarrow$ 3-way Handshake (Agreement)
L4 ensures packets getting delivered in-order.
↳ Transmission Rate

- TCP Header :
  1. Sequence number : Allows sequencing of data
  2. Acknowledgement number : Sent back from reciever, to let the sender know that data was recieved and is waiting for next data.

Data is sent $\longrightarrow$ Wait to recieve acknowledgement (OS starts a timer)
↙ Data is recieved ↘ Data is lost
↓
Retransmit the data

OS does not want to waste resources
$\Rightarrow$ $\therefore$ Starts a timer and if it elapses $\Rightarrow$ Timed Out

Data can be lost even when acknowledgement no. > sequence no.

Acknowledgement cache $\longrightarrow$ Prevents $\underbrace{\text{Deadlock}}$
↓
Sender sends & waits for response (ACK)
Reciever sends back & still on the way

" IIIT Timer $\longrightarrow$ 1½ hour delay "

- RTT : Round Trip Time
  Sample RTT : How much time
  Estimated RTT : Keep updating estimation on every sample
  $(1-\alpha)$ ERRT + (α) SRTT
  ↳ If very high, $\therefore$ Works only on current & not past
  Default : 0.125

DeviationRTT : Worst case scenario, can happen but rarely

$$DRTT = (1-\beta)\, DRTT + \beta \times | SRTT - ERTT |$$

↳ Weighted moving range ($\beta : 0.75$)

Timeout Interval $= ERTT + (4 \times DRTT)$

- 3 → 85 % confidence
- 2 → 65 % confidence
- 4 → 99 % confidence

[Based on experiments]

Q) Check acknowledgement constantly for every packet ?

Disadvantage (if check) ⇒ High overhead & Latency

Solution ⇒ Delayed Acknowledgement

↳ Wait after sending multiple packets and reciever sends ACK

To prevent deadlocks :

Timer on the reciever end

③ Window : Flow Control

Reciever has a limit to recieve data & then waiting time.

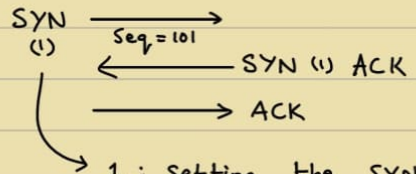
Send a 0 byte data to know whether reciever can recieve data.

Reciever & Sender ⟶ Both send data, but have seperate values of ACK.

Sequence number → Any random number within the bounds.

↳ Decided after agreement

3-way handshake ⟶ "Can we start connecting ?"



↳ 1 : Setting the SYN bit, i.e. sending the acknowledgement

Agreement on Seq. no. from both sides

ACK → 1 : Accepts the mutual seq. no.

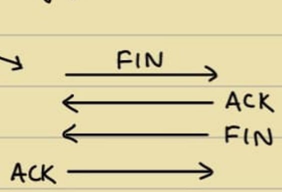
SYN ACK : Mutual Agreement & that both recieve data

SYN : 1 ⇒ Connection establishes and sender initiates sending

Closing connection : FIN and RST flags

FIN → Mutual way termination (4 - way exchange)

RST → Forceful way termination
No acknowledgement ∵ connection is terminated



Note : Size of TCP Header : 30 bits to 60 bits (Option Speed)

P bit : [Push bit]
Send data ⟹ Reciever gets and puts in buffer (Generally)
P = 1 ⇒ Data is directly sent to process , No buffer used

U bit : [Urgent bit]
Some packets sent need to be processed urgently and rest can be processed later

- Memory :
  Every Process requires memory.
  Memory Virtualization ⟶ Every process feels like its getting entire memory.

  Some of the RAM is used up by OS , code and data . Rest of the memory is usable RAM , i.e. our code and data.

Multiple Processes running at the same time ⇒ Fast

Ways to achieve process virtualisation :
(1) Partitioning OS into slots
Disadvantages : - Other slots data can be accessed.
- Each process may not be fixed amount of data.

{ Saturday : 8:30 Tutorial ⟶ Class }

4/9

Flow control : Just b/w sender and reciever (L4)

Congestion control : C-bit , In the network layer

- Memory virtualization :

  e.g. Supermarket vs e-shopping

$$\begin{tikzcd} \text{Every process} \arrow[r, leftarrow] \arrow[dr, bend right] & \text{Address space} \\ & \text{Virtual Address Space (VA), mapped to Physical memory} \end{tikzcd}$$

$*p = 3 \Rightarrow \&p \leftarrow$ Virtual memory

Note : Only 1 Physical Memory $\longleftarrow$ Physical Address (PA)

Processes might not always stay in Physical memory, it can be in disk as well from overflowing. (Note : Swap Space)

$*$ Translation : Memory Management Unit (MMU)

$$\begin{tikzcd} \text{Translation} \arrow[r] \arrow[dr, bend right] & \text{Fast} \\ & \text{Conversion + Fetch} \end{tikzcd}$$

- Goals of Virtualization :
  - (1) Transparancy $\leftarrow$ Each process gets an illusion of infinite space
  - (2) Efficiency $\leftarrow$ Use hardware
  - (3) Protection

- For Process Virtualization :
  - Mechanism : Limited Directed Execution (LDE)
  - Policies : Scheduling Algorithms

- For Memory Virtualization :

  Assumptions :

  (1) User address space is contiguous in Physical Memory.

  Process $\longrightarrow$ Block

  (2) Size of Address space is too big, less than size of Physical memory

  Process $\longrightarrow$ No overflow

  (3) Every Address space is of equal size.

Memory Virtualisation :

Q) How do I divide Physical Address space.?

Q) How do I map Virtual Address to Physical Address ?

Reference the variable ← Stack

Note : $\underbrace{\text{Base}}$ and $\underbrace{\text{Bound}}$ approach : → Values are stored in MMU

Then, Add Base Address

VA ⟶ Process starts at 0

But PA : Process = 0 + Base value

= VA + Base

Accessing after bound ⟶ Fault : Address out-of-bound error

Allocation : Dynamic ⟹ Dynamic Relocation

VA to PA : $\underbrace{\text{Address Translation}}$ → MMU not OS

* $\underbrace{\text{MMU}}$ ⟶ @ Context-Switch

↪ Has only 1 pair of Base and Bound

* Segmentation : (Memory Management System)

Code , Stack & Heap – Generalised Base and Bounds

∀ Segment : Has a base and bound , Not Fixed

→ Dividing PA space

Segment Registers in MMU .

First ⟶ Identify which segment

* Address Translation :

Stack ⟶ goes up ∴ Subtract

Real Address (PA) = VA + base address

Code : No offset

Heap : Offset

Calculation

6/9

VA → Every Process

Map : $\underbrace{\text{VA}}_{\text{0 to Max}}$ ⟶ PA : Base and Bounds ⟶ Disadvantages :
(1) Fragmentation
(2) Unused Memory

- Identify segment :
  (1) Implicit :
  Stack
  (2) Explicit : (VA → 14 bits)
  First 2 bits ⟶ Segment (12-13)
  Rest ⟶ offset (0-11)

  00 : Code
  01 : Heap
  11 : Stack

∵ Bits are unused ⇒ Code and Heap have same bit in some OS.

- <u>Simple Address Translation</u> :

  Many OS : Segmentation ✗
  Paging ✓

  PA : Base + $\underbrace{\text{Offset}}_{\text{Data - Max stack data}}$

- $$\begin{tikzcd}
& \text{Segmentation} \arrow[dl, no head] \arrow[dr, no head] & \\
\text{Fine-grained} & & \text{Course-grained}
\end{tikzcd}$$

  Fine-grained
  ├ Small size segments
  └ High overhead

  Course-grained
  ├ Large size segments
  └

When different sized segments, Disadvantage :
External Fragmentation ⟵ Extra memory is available
But not contiguous, i.e. has holes

Defragmentation : High overhead

Note : Free space management algorithm, Closest Fit, First fit are algorithms

Fixed size segments $\Rightarrow$ Code, Heap & Stack ×

↳ Disadvantage : $\underbrace{\text{Internal Fragmentation}}$

↳ Better than external fragmentation

Page : Fixed size segment in a process

Paging : VA and PA are both divided

$\therefore$ If 4 pages $\rightarrow 2^{②}$: 2 bits

64 bytes VA $\rightarrow 2^{⑥}$: $\underbrace{\text{6 bits}}$

$\underbrace{-\,-}_{\text{Page}}\underbrace{-\,-\,-\,-}_{\text{Location in page}}$

VPN : Virtual Page Number

PFN : Page Frame Number } ← Every page has a number

In PA : Page Frame

In VA : Page

Advantage :

(1) Flexibility : OS finds a free page

(2) Simplicity : $\forall$ Process , $\exists$ $\underbrace{\text{Page table}}$

↳ Mapping b/w VA and PA

Array can be used , e.g. $VPN[i] = PFN_j$

Note : Multiple VPNs can be mapped to a PFN (not same process)

If $Mem_{Process} > Mem_{RAM} \Rightarrow$ Swap in & Swap out from Hard disk

- <u>Process of Translation :</u>

VPN		Offset			
VA_5	VA_4	VA_3	VA_2	VA_1	VA_0

e.g. mov 21 %eax

$(21)_{10} = (10101)_2 = (010101)_2$

01 : VA space
0101 : offset } $\rightarrow$ & if $VPN_1 = PFN_7$ (from page table)

Then $\underbrace{010101}_{VA} \Rightarrow \underbrace{1110101}_{PA}$

$(1110101)_2 = (117)_{10}$

Page Table Base Register $\longrightarrow$ store base of the address of the page table.

e.g. 32 bit address space with 4 KB pages
no. of bits for offset ?

$4\text{ KB} = 2^{12} \Rightarrow$ 12 bits to represent
$32 - 12 = 20$ bits for mapping, i.e. VPN
$\therefore 2^{20}$ mappings per process
Each mapping $\rightarrow$ 4 bytes
$\therefore 2^{20} \times 4 = 4$ MB per process per page

If 100 processes $\Rightarrow \underbrace{400\text{ MB}}$ for address translation
$\hookrightarrow$ Page Table
$\therefore$ Use pages to store page tables

- <u>Page Table</u> :
Page Table Entry (PTE)
VPN , PFN , V, P, $\underbrace{\text{Present}}$, $\underbrace{\text{Dirty}}$, $\underbrace{\text{Ref}}$

$\hookrightarrow$ If page is recently used
$\hookrightarrow$ If page is modified (after from memory)
$\rightarrow$ If page in Physical memory or secondary
$\rightarrow$ Protection Bit
Valid Bit

$\therefore$ Disadvantage : Fetch twice for every $\underbrace{\text{translation}}$
(Extract memory)

$$\begin{tikzcd} \text{PTBR} \arrow[d] \\ \text{Page Table} \arrow[d] \\ \text{PTE} \arrow[d] \\ \text{Find} \end{tikzcd}$$

- Efficient Translation : Caching
Translation Lookaside Buffer (TLB) - Register
$\hookrightarrow$ Address Translation cache
$\hookrightarrow$ In MMU

9/9

TLB → Hit rate of cache ↑

Cache : VPN ⟷ PFN

Size of cache ↑ ⇒ Hit rate ↑

Size of page ↑ ⇒ Hit rate ↑

- Spatial locality : Space dimension , i.e. same space accessed
- Temporal locality : Time dimension , i.e. recently accessed.

- Hardware TLB handling - Hardware goes through the page in O(1).
  Software TLB handling - Hardware raises exception (TRAP)
  Context Switch from Hardware to Software

@ TLB miss handlers → stored in OS Kernel, avoiding translation (Physical memory)

TLB → Fully Associative cache
(No direct mapping , random)

- TLB :
  - Valid bit :
    Translation is valid or not → Page Table Entry case
    Process A to B ⇒ B should not use A's mapping → TLB case
    ∴ A → invalid

  - ASID :
    Every process has a Address Space Identifier .
    Mapping : $\underbrace{\text{Process id}} \leftrightarrow \underbrace{\text{ASID}}$
    ↳ Small (∵ Cache)
    ↳ Disadvantage : Large amount of bits (32 to 64)

- (i) Segmentation + Paging } → Reduce Page Table size
  (ii) Encoding

Page size ↑ ⇒ Mapping ↓ but Internal Fragmentation , lot of empty sizes will be left

(i) $\underbrace{\text{Segmentation}}$ + $\underbrace{\text{Paging}}$ → Getting best of both ?

↳ External Fragmentation ↳ Internal Fragmentation } Issues

$\underbrace{\text{Page Table}}$ → stored in $\underbrace{\text{RAM}}$
↪ divided into pages
↪ Stored using Pages

Page Table contains mappings, entries.
Each mapping is not important, only valid entries are
Invalid entries → don't store, store only when it becomes valid.
Valid entries → store
∴ Multi - Page Table (Tree - like)
i.e. Meta-index, multiple pointers heirarchy.

*[Explore 2-level Page Table
Study 3-Level Page Table]*

Page Directory → Data structure for multi-level page tables.

PDBR → less space (∵ Only valid PTEs are stored)

Disadvantage : Time overhead

12/9

- Inverted Page Table : 1 page table per process
  disadvantage : Linear scan is expensive

- OS creates Memory Heirarchy.
  Physical Memory → Faster access
  Less capacity
  Disk → Slower access
  More capacity

On-demand Pages
- Swap Space :
  ↪ Allocated in $\underbrace{\text{disk}}$
  ↪ divided into blocks ≈ Page size

- Present bit :
  1 : page is in the page table
  0 : page is in the swap space

- Valid bit = 0 : Page is not in Physical memory $\Rightarrow$ In disk

  $$\text{AKA } \underbrace{\text{Page Fault}}_{\text{Process moved to block state}} \longrightarrow \underbrace{\text{Handler}}_{\text{What to swap, when \& from where AKA Page Replacement}}$$

- AMAT
- Optimal Replacement Policy (Theoretical Optimal)

  Belady Replacement Policy (But Not Practical)

  ↪ Possible future access

Reduce any policy to Belady for comparison

(i) FIFO → First in First Out

Cache size ↑ $\Rightarrow$ no. of hits ↑ (Belady's Anamoly)

↪ Not always, depends on ordering of stream of data.

(ii) LRU → Least Recently used

- Valid bit = 1
- Access bit = 1

  ↓ Recently modified

  Access bit = 0 ⟵ Evict a page

- Thrashing → Excessive Page fault Handlers

{
Sat → Tut (IMP) [Bonus Quiz]
Tues → Tut (Attendance) [MP2]
Fri → None
}

26/6

SMTP uses TCP
(Spotify – Web series)

Install Apache / ngnix webserver

→ **Peer 2 Peer Network :**

Spotify : Midserver + Peer2Peer
(Client server)

→ **Application level Protocol :**

HTTP : 80

HTTPS : 443 (HTTP over secure network)

$\forall$ connection → 3 way Handshake ($\because$ $\underbrace{\text{HTTP}}_{}$ runs on TCP)
↳ Stateless

- 1.0 HTTP ← For fetching every connection, connection is established.

  1.1 HTTP ← Maintain connection to fetch all objects and then terminates.

  2.0 HTTP & 3.0 HTTP

→ **Cookies :**

$\because$ HTTP is stateless, Data stored in browser cache.
Metadata is stored in server.

Web cache :

Browser cache
ISP cache
Regional cache
} If recently changed, use cache
Main server

- Traditionally,
  Client ⟷ Server : Takes time
  (Delay)

  $\therefore$ Add cache at server.

Content distribution Networks (CDN) :
↳ Servers distributed among diff. locations.
(Logics : Potential viewership + location)

Enter Deep : Build smaller clusters in more sites
Bring Home : Build large cluster in lesser sites

Cache does not contain originals
↳ Conditional GET : Data from server would only be header, no body

HTTP 2.0 → Current
HTTP 3.0 → Future, but has support now.

→ Domain Name System (DNS) :
Translation b/w name and I/P address. Allows aliasing.
Uses UDP.
- UDP call to get I/P address & HTTP call happens, i.e. then client uses HTTP to send request to server.
- DNS record
- BIND → Turns machine to DNS server → Port 53
  i.e. machine acts like a directory, returns IP address
- Any request (HTTP, SMTP, etc) sent translates the hostname to IP address using DNS

$\underbrace{\text{1. Root DNS}}_{\text{12 different Organizations are responsible for DNS creation, etc}}$

$\underbrace{\text{12 different Organizations}}_{}$ → Report to IANA

- 1. Root DNS
  2. Top-level Domain (TLD) : .com, .org, .ai domains
  3. Authoritative Domain

- How it works : (Recursive Process), When you send a request to a host,
  1. Request is sent through Port 53
  2. Local DNS (ISP Provides) ← Cache kind-of
     If mapping exists, fetch. Else goes to next level (Recursion)
  3. Root DNS (if it has, it will give back, else send to next level)
  4. TLD DNS (if it has, it will give back, else send to next level)
  5. Authoritative DNS (Then, get I/P address and return to client)
  6. Then, you send a HTTP request to the server.

- DNS servers stores Resource Records (RR) : $\underbrace{\text{(name, value, type, ttl)}}$
  ↙ tuple ↳ Each DNS record

ttl : time after which it needs to be updated from the root DNS.
↳ Time To Live

(i) A type DNS record : (Authoritative)

Name : Hostname

Value : I/P address

$\left(\begin{array}{l} \text{AAA} \longrightarrow \text{IPV6} \\ \text{A} \longrightarrow \text{IPV4} \end{array} \right)$

(ii) CNAME type ⟶ Mapping one domain to another domain

↳ Canonical name record (alias)

(iii) NS ⟶ Named Server

Main server itself (Authoritative)

(iv) MX type ⟶ Mail

(mail.outlook.com)

27/9

- Concurrency :

Multiple execution points sharing the same memory space

Same process – Multiple threads sharing same memory space.
- The threads synchronize with each other.
- Should not overlap ideally.

Thread : Smallest unit of execution.
Seperate P.C
Seperate stack of local variables
Creates child processes
Same address space / memory

Concurrency in Single core machine ⟶ Context switching

fork ⟶ child is created, PID diff. from that of parent.
Memory copy is created, code is shared.

exec ⟶ Child is created with same PID.
Replace the memory, ∴ diff. memory
Inter-process communication ⟶ reading / writing from shared files

Multiprocessing : Multiple processes executing in diff. cores of the CPU in the same time on the same data.

Scheduling → Thread level
Not Process level

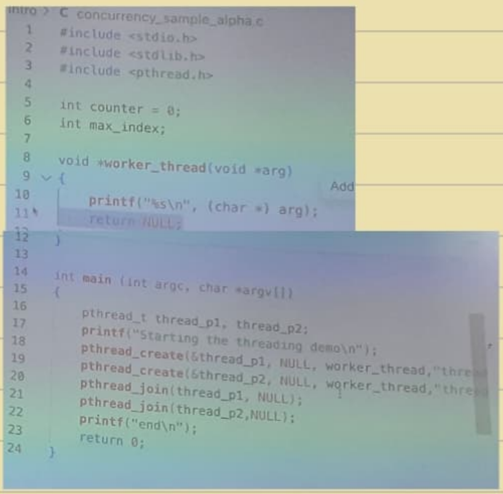
- Kernel Space → Kernel level threads → Scheduled by Kernel
  User level threads → Scheduled by user, i.e. user level libraries ← pthreads

- Concurrency : Interleaving
  - Context - switching in case of single core
  - Dealing with lot of things at once.

  Parallelism : At same point of time → Multiple execution
  ∴ Multiple cores are required.
  - Doing lot of things at once
  - Subset of concurrency

- TCB : For every thread, ∃ TCB
  Kernel level threads : Scheduled by OS, Handles system calls.
  User threads : What process a developer writes, e.g. processing a file.

- Start routine

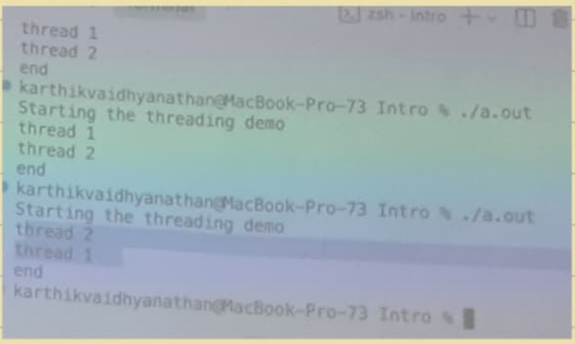


```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int counter = 0;
int max_index;

void *worker_thread(void *arg)
{
    printf("%s\n", (char *) arg);
    return NULL;
}

int main (int argc, char *argv[])
{
    pthread_t thread_p1, thread_p2;
    printf("Starting the threading demo\n");
    pthread_create(&thread_p1, NULL, worker_thread,"thre
    pthread_create(&thread_p2, NULL, worker_thread,"thre
    pthread_join(thread_p1, NULL);
    pthread_join(thread_p2,NULL);
    printf("end\n");
    return 0;
}
```



```
thread 1
thread 2
end
karthikvaidhyanathan@MacBook-Pro-73 Intro % ./a.out
Starting the threading demo
thread 1
thread 2
end
karthikvaidhyanathan@MacBook-Pro-73 Intro % ./a.out
Starting the threading demo
thread 2
thread 1
end
karthikvaidhyanathan@MacBook-Pro-73 Intro %
```

↳ Non - deterministic + No wait()
Depends on scheduler

Reason : Race condition
↳ access b/w shared variables

No synchronisation