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;
}
```

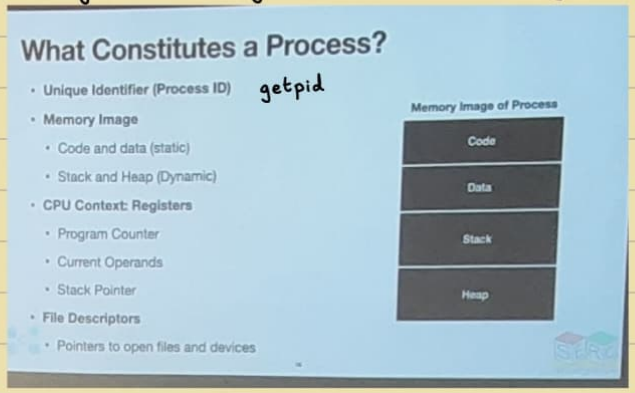
```c
    }
    printf("Who am I? PID:%d\n",(int)getpid());
    return 0;
}

// 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]

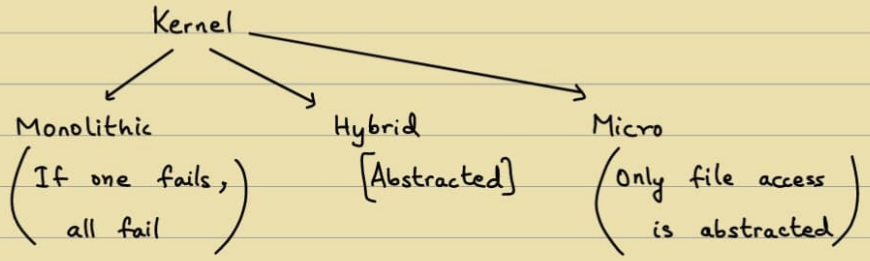


Create memory image for the process

⇓

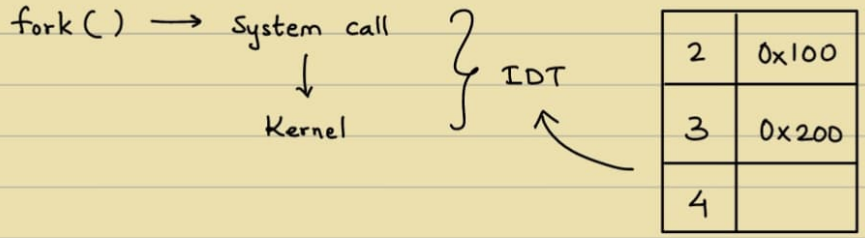
Allocate PC, SP

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



Kernel has stack
↳ Process

- API → System calls (fork, exec, wait)
  - ↳ Resides in Kernel
    - Limited Direct Execution (LDE) → CPU



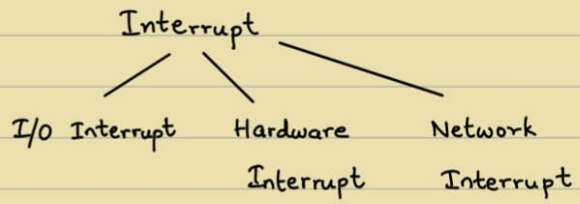
System call ⇒ Mode change ⇒ User mode to Kernel Mode
↳ TRAP Instruction : Changes privilege

IDT ⇒ OS code gets executed
↳ Interrupt disrupt Table

Context → PC, SP → Saved

User stack ✗
Kernel stack ✓
↳ Per process level

```
10
11  syscall  <- OS
12
```

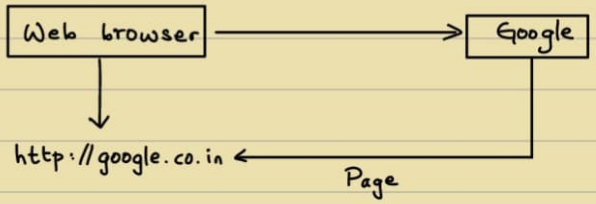


- Context Switch:

  How to decide which process to run next ⟶ Scheduling

  In reality, Jobs can arrive at any time.

26/8



Port → 80

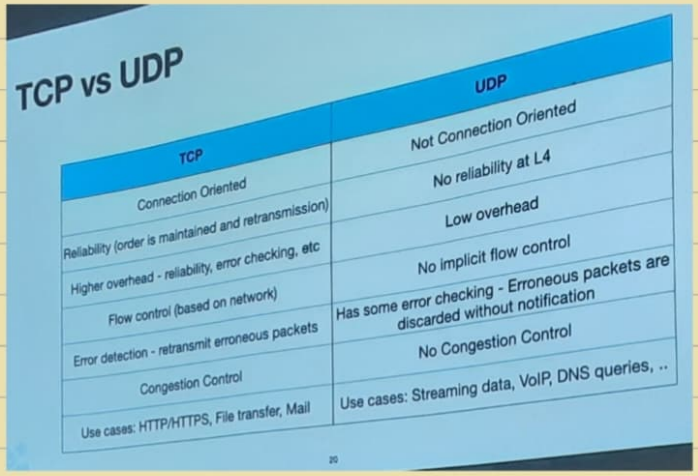
Process id	Port

← 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)



Flow control : Control speed of data transmission.

2/9

- UDP ⟶ Unreliable at L4 ∵ Fire and forget, i.e. responsible for giving data but not ensuring recieving.

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

TCP ⟶ 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 ⟶ Wait to recieve acknowledgement (OS starts a timer)

Data is recieved
Data is lost ↓ Retransmit the data

OS does not want to waste resources
⇒ ∴ Starts a timer and if it elapses ⇒ Timed Out

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

Acknowledgement cache ⟶ Prevents Deadlock
↓
Sender sends & waits for response (ACK)
Reciever sends back & still on the way

"IIIT Timer ⟶ 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, ∴ 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 ($\beta : 0.75$)

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

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

[Based on experiments]

② 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.

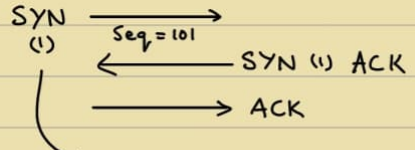
Send a 0 byte data to know whether reciever can recieve data.

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

Mutual way termination (4-way exchange)

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

  Every process ⟵ Address space
  ⟶ Virtual Address Space (VA) , mapped to Physical memory

  *p = 3 ⟹ &p ⟵ Virtual memory

  Note : Only 1 Physical Memory ⟵ 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)
    ⟶ Fast
    ⟶ Conversion + Fetch

- Goals of Virtualization :
  (1) Transparancy ⟵ Each process gets an illusion of infinite space
  (2) Efficiency ⟵ 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 ⟶ Block
  (2) Size of Address space is too big , less than size of Physical memory
  Process ⟶ 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 : Base and 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 : Address Translation

→ MMU not OS

* 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 : VA → PA : Base and Bounds → Disadvantages :

VA → 0 to Max

(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.

- Simple Address Translation :

Many OS : Segmentation ✗

Paging ✓

PA : Base + Offset

Offset → Data − Max stack data

-

Segmentation

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 ⟹ Code, Heap & Stack ×

↳ Disadvantage : Internal Fragmentation

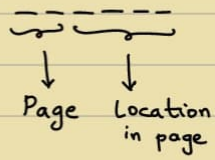
↳ Better than external fragmentation

Page : Fixed size segment in a process

Paging : VA and PA are both divided

∴ If 4 pages → $2^{②}$: 2 bits

64 bytes VA → $2^{⑥}$: 6 bits



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 : ∀ Process , ∃ 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 $Mem_{Process} > Mem_{RAM}$ ⟹ Swap in & Swap out from Hard disk

- Process of Translation :

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 } → & if $VPN_1 = PFN_7$ (from page table)

Then $\underbrace{010101}_{VA} \Rightarrow \underbrace{111\,0101}_{PA}$

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

Page Table Base Register ⟶ store base of the address of the page table.

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

$4\ KB = 2^{12} \Rightarrow$ 12 bits to represent

$32 - 12 = 20$ bits for mapping, i.e. VPN

$\therefore 2^{20}$ mappings per process

Each mapping ⟶ 4 bytes

$\therefore 2^{20} \times 4 = 4$ MB per process per page

If 100 processes $\Rightarrow$ 400 MB for address translation

→ Page Table

$\therefore$ Use pages to store page tables

- Page Table :

Page Table Entry (PTE)

VPN , PFN , V , P , Present , Dirty , Ref

- V → Valid Bit
- P → Protection Bit
- Present → If page in Physical memory or secondary
- Dirty → If page is modified (from after memory)
- Ref → If page is recently used

$\therefore$ Disadvantage : Fetch twice for every translation
(Extract memory)

- Efficient Translation : Caching

Translation lookaside Buffer (TLB) – Register

→ Address Translation cache

→ In MMU

PTBR
↓
Page Table
↓
PTE
↓
Find

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 : Process id ⟷ ASID
    ASID → Small (∵ Cache)
    Process id → 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) Segmentation + Paging ⟶ Getting best of both ?

Segmentation → External Fragmentation
Paging → Internal Fragmentation
} Issues

Page Table → stored in 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 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 ⇒ In disk

AKA Page Fault → Handler

Page Fault → Process moved to block state

Handler → 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 ↑ ⇒ 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