Date:

Operating Systems and Networks

OS is the middleware blw hardware and the application/user programs. It provides abstraction to the users for various things like the memory, context switching, storage, and other HID

Virtualization :- Processes have the illusion of it being the only process with access to its own unique memory.

Concurrency :- Multiple processes can run simultaneously

Persistance :- OS also handles interactions with the disk and performs storage management

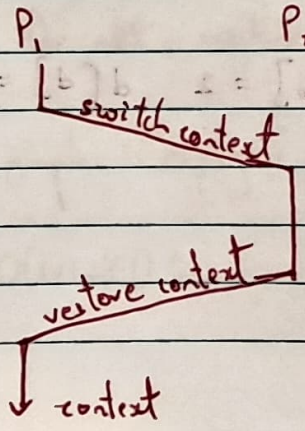
Process Virtualization :-

Process :- Running ~~programs~~ → set of instructions

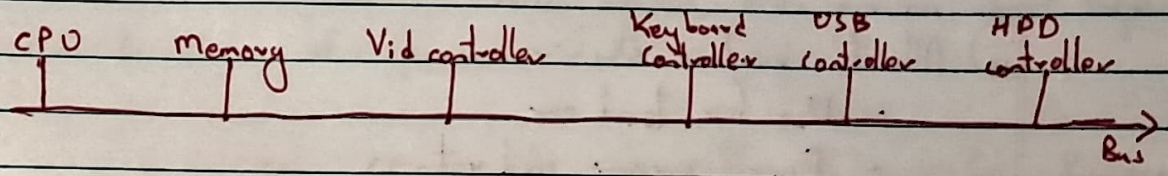
Program $\xrightarrow{\text{Compile and build}}$ Executable $\xrightarrow{\text{Execute}}$ Running Process

Only running ~~process~~ programs ~~consume~~ resources.

Each process has its own CPU. Multiple processes can run on the same CPU with the help of context switching (virtualization of the CPU)



We need something to store context, some idea of the requirements to know which process to switch to.



The farther we are from the CPU, the longer it takes.

Usual execution pipeline:- Fetch - Decode - Execute. [but usually superscalar]

Context:- Registers values, Stack pointer, stat codes, PC next

We need both low level mechanisms and policies to successfully context switch.

Constituents of a process:-
Memory (Static & Dynamic), Instructions, data
stack heap

Memory image, state and address space

Memory:- (Accesible) → address space
Instr ptr, pc :- which instr is exec
Stack ptr :- local vars, fns and ret addr
Persistent storage :- I/O info
Unique Identifier :- Proc id.
File descriptors :- Ptrs to open files and devices.

Mem Img - Code, data, stack, heap

Proc creation:-

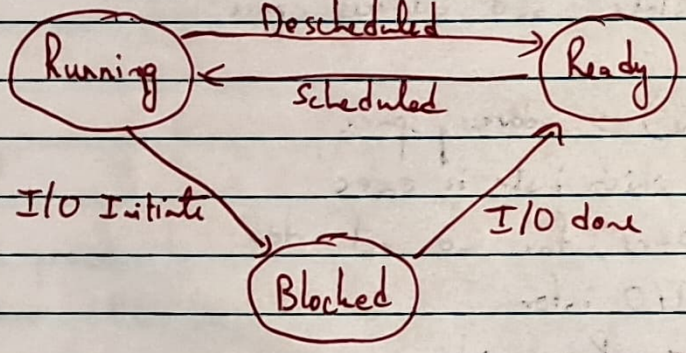
(i) Load program into memory
- Init prog on disk
- OS does Lazy Loading → Load only relevant stuff

(ii) Allocate runtime stack:-
- Use for local var
- Func params and returns

(iii) Creation of Prog heap:-
- Used for dynamic data like malloc and free

(iv) Basic file setup
- Stdin, Stdout, Stderr

State of a process:- Multiple states like "running", "suspended", "blocked", "ready"
- "blocked" → waiting for I/O
- "ready" → due to context switching



The processes are stored in a list called process list
Process Control Blocks are stored in the list. (structs)
- Proc id
- State
- Address Space

Kernel:- Part of the OS that handles direct access to the hardware.
Monolithic, Hybrid and Micro.

The OS should be able to create a process, kill a process, ~~wait~~, make process wait, suspend or find the status of a process.

APIs allow different programs to communicate with each other. Provides a software interface

System calls are methods by which the OS provides some functions that can be leveraged by user programs.

POSIX (Portable Operating Systems Interface)
→ Standard set of system calls that an OS must implement

This ensures uniformity for languages to not worry about system calls syntax

There are two modes of execution :- User mode and Kernel mode
normal access for the user ↓ Higher privilege

The first process that is run on boot is the init process. It is the ancestor of all processes

Fork:- A new independent child process is created. Execution of two happens simultaneously.

Wait:- calls blocks/bubbles in parent until child terminates
If proc terminates in the absence of wait -> zombie process. Init adopts orphans and reaps them.

KV has doll-dole merger, but in ... direction

Exec:- Load a different executable to the memory. Fork & exec are used in conjunction to retain control.

We need to ensure that processes do not ~~do any thing~~ some thing unexpected, and also stop and switch b/w process.

Hardware Support:- Have some low level mechanisms.

Limited Direction Execution:- Every process gets some time to run on the CPU, after which the OS shifts the context to another process

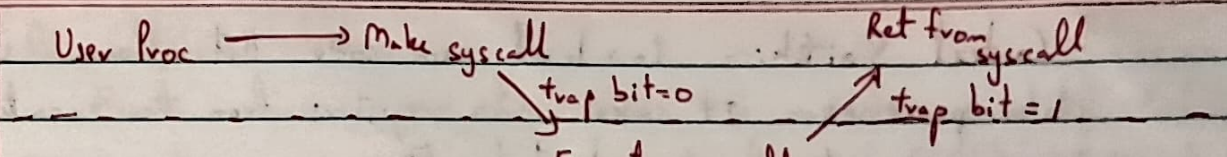
Interrupt Descriptor Table:- Mapping b/w id and actual location of the system calls. Useful in switching from User to Kernel mode, as it obfuscates the true location of syscalls.

Trap instr:- Special kind of instr to switch from user to kernel mode. It raises the privilege from User to Kernel mode. Return-from-trap allows switching back into user mode and returns to the calling program. The normal routine is interrupted. Context must be saved in the Kernel stack. The OS has multiple kernel stacks, one per process.

→ CPU goes to higher privilege [Kernel Stack]
→ Switch to kernel stack and save context
→ Look up in IDT and jump to Trap handler
→ Perform privileged instrs.
→ Call return-from-trap
→ Return to User Mode.

We use Kernel Stack for safety reasons.

User Mode



Kernel Mode

Interrupt:- Signal that conveys that "some thing" has happened Due to unexpected events. I/O interrupt, Console Interrupt, etc. Can be Hardware or Software.

Trap:- Software interrupt caused by exceptions or syscalls.

The trap instr is called by the syscall.

Switching blw Processes

(i) Cooperative [Non-preemption Approach]:-
→ Wait for system calls
→ Process transfers the control to the CPU by making a sys call
→ There can be misbehaving processes → divide by zero, etc. We the Trap to OS → Process gets terminated
→ Inf loops sans termination are issues → Reboot! for getting control.

(ii) Non-cooperative [Preemptive Approach]:-
→ OS gains control by using interrupts.
→ Timer interrupt → Every x time units, halt process, invoke interrupt handler and OS can choose to change the process. OS starts timer on boot
→ The CPU with the help of Scheduler chooses to continue current process post interrupt
(or)
Perform context switch

Date:

Context Switch:- low level asm code that allows switching b/w processes. It saves a few registers from the executing proc regs to the kernel stack.

What if during handling of one interrupt, another interrupt occurs?
Disable other interrupts!
Create sophisticated locking mechanisms to safeguard that block of memory from concurrent access.

Scheduling ~~Processes~~ Policies:-

To evaluate the quality of a scheduler, we need estimates like the no. of proc in the queue, what is their goal/purpose, how much time do they require etc, I/O reqs, Mem reqs etc.

Each process that is in the queue in execution → Job

Assumptions:-
→ All processes runs for the same time
→ Arrive at same time
→ Use only CPU
→ Run time is known.

Performance Metric:- Turnaround time
$T_{turnaround} = T_{completion} - T_{arrival}$

Fairness:- Guarantee that all processes at least gets the chance to execute.

Fairness & Performance may not go hand-in-hand

All assumptions intact:-

Assume 3 proc arrive at the same time and have same estimated run time. We just pick a process (rand or FCFS) and service. But if we relax the assumption that all proc take same time, FCFS is not a great idea. (convoy effect)

We then choose to schedule the shortest job first. [SJF]

But this is problematic if we the processes can arrive at various times. So what we do is when a new job arrives, we schedule the job with the Shortest Time to Completion First
↳ Preemptive

Perf Metrics:- Response Time

$T_{response} = T_{firstrun} - T_{arrival}$

LLMs stream the output instead of sending it in one chunk just so that the user feels that the response time is minimal

A better idea is to run all process for a short time and the preempt. Response time is optimised, but Turnaround Time is bad.

↓

Round Robin:- Run jobs for a time slice → switch to next job → repeat till all are done.
Time slice is the short time for which a process is run. Uses timer interrupts. There is an Overhead of Context Switch.

Tradeoff b/w Turnaround time and Response Time
Turnaround only cares about completion while Response Time cares only about performance / interactiveness / ping.

Round Robin (naive) doesn't work well with I/O

During I/O, the process goes into a blocked state and then waits on interrupt.

We can instead leverage STCF along with RR by executing other processes while one process is performing I/O

What if we do not know the length or expected time of completion of the job? (We usually don't)

So let us prioritize?

Multi Level Feedback Queues

→ Reduce turnaround times by running shortest job first

Use n-queues for n-different priority levels.
Use the priority to decide which job to run
Job with higher priority ⇌ Job on a higher queue.

Scheduler keeps updating priority based on observed behaviours.

One of the metrics used is the time taken for execution within the time-slice allocated.

→ Interactive jobs ⇒ Higher priority - (Keep returning to CPU)
→ CPU intensive ⇒ Reduce priority

→ If Priority (A) > Priority (B) → Run A
→ Else if P(A) = P(B) → RR A & B

R R R

Every new job is placed at high priority.
- ☹ If job uses entire time slice, reduce priority
- ☺ Elif keep at same priority.

Incorporating I/O:

Assume there is a highly interactive job that relinquishes CPU faster s.t the priority does not change.
Long term jobs suffer.

Starvation: Long running jobs never get CPU time if there are many interactive jobs

Game the Scheduler: Processes can game the scheduler to get more than what is fair, by using I/O to relinquish the CPU before time slice is over. This results in the priority of the process not changing.

Programs may also change behaviour over time. There might be CPU intensive phases b/w phases of interactiveness.

Priority Boost!: Every x seconds or so, bring all processes to the highest priority queue or increment their priorities by 1. This ensures that there is no starvation.
It also takes care of changed behaviours.

To reduce gamification of the scheduler, we perform reverse priority boost by pushing all high priority processes into lower priorities

Too small S → interactive proc doesn't get enough CPU
Too large S → starvation.

We have have different time slices for different priority queues

We're doing Networks ?!

At the most basic level, networks are hardware (cables)

The process raises an interrupt, asking the OS to transmit the data to another system. The hardware at the other system raises an interrupt and lets the OS access the data.

We need interfaces b/w process & network, and b/w network components.

Host:- Any device that can send or receive traffic.
Hosts can be client or server.
The roles are not ME, each can be the other.

Client:- Any host requesting for resources on the network.
Server:- Any host providing resources on the network.

Hop:- A single traversal

We need the **IP address** (→ Public & Private) to request a server.
↳ Gives the location of a point on the network.

When client sends data, it sends both source and destination IP

Repeater:- Repeaters repeat signals to increase signal strength
Floods the network

Hub / Bridge:- ~~Hubs~~ Multiport repeaters. Broadcasts the data
→ Intelligent hubs. Lets the hosts know what hosts exist on the other side, and ~~sends~~ data to the correct host.
broadcasts → Still floods while we communicate from one side to the other.

Switches :- More intelligent bridges. unicast data rightly. Have switching tables.

Router :- Used to connect blw networks. They route the data to the correct network. Routers store all routes in a routing table. They provide security, filtering and redirection. They have IP addresses and specific ones pertaining to the connected networks.

Access Point :- Wireless switches

Port no.s are associated with processes

Networking Layers :-
Reduces complexity and provides abstraction.

OSI model:
Application → Provides network services to applications.
Presentation → Ensures the data is in an understandable format
Session → Manages connection blw diff devices
Transport
Network
Data Link
Physical → Data in the form of 0/1. Smth has to transport bits from one layer to another over physical media.

Data Link → Hop to Hop connectivity. Responsible for creating link blw two directly connected nodes, error correction. MAC addresses are used here.

Network Layer → Manages routing through routers in a large network. Uses IP addresses to perform fns like Logical Addressing, Path selection and Packet Switching.

Transport Layer:- Ensures data transfer is reliable, sequenced, manages flow control and error correction. Uses TCP/UDP.

4 Layer Model:
- Application - HTTP, SMTP, RTP, DNS
- Transport - TCP, UDP.
- Internet - IP, ICMP
- Network/Link Layer. - DSL, Ethernet, 802.11, SONET

Transport Layer:-

Service to Service Delivery:-
(i) MUXing & DEMUXing
(ii) The addressing scheme is Ports
(iii) TCP → Transport Control Protocol → Reliable
UDP → User Datagram Protocol. → Efficient

→ Client sends a request to the server
→ The server provides a reply based on the request made by client

MUXing →
Software component in the OS that supports network calls is the Protocol Stack

Drivers are the interface b/w hardware and the OS

Socket is the abstraction provided to a process by the OS to connect to the network
The network service API is used to write all network applications.

Streams:- Reliable (connection oriented)
Datagrams:- Unreliable (connection less)

Allows applications to attach to the network at different ports.

Socket API

sock socket() :- Creates a new socket of a certain type and returns file descriptor

bind():- Associates the socket with an IP & port

listen():- For server sockets, allows listening for incoming connections

accept():- For server sockets, it waits for client to connect & ret the new f.d.

connect():- For client sockets to create a connection with the server

send()/receive():- Transmit data

close():- Close the connection

Appln process is identified by the triple (IP, Protocol, Port)
16-bit integers ↵
that process leaves.

Servers bind to well known ports
Clients are assigned ephemeral ports that are chosen by the OS temporarily.

Types of links:-

(i) Full Duplex:- Bidirectional Simultaneously

(ii) Half Duplex:- Bidirectional sequentially

(iii) Simplex:- Unidirectional

Segment carry data across the network, within packets within frames

L4 : HTTP/SMTP — Segment + IP — Packet + Frame

MUXing:- Handle data from multiple sockets, add transport header
DeMUXing:- Use header info to deliver received segments to the correct Socket.

DeMUXing:-
Host receives IP datagrams
Each datagram has a source & dest IP
Carries one transport layer segment
Each a segment has source & dest port no.
IP addresses and ports are used to direct the data.

Connection Oriented (TCP):-
(i) TCP Socket identified by a 4 tuple :- Source IP, dest IP, source port & dest port.
(ii) Receiver uses all 4 to direct segments to appropriate socket.
(iii) Server may support many TCP sockets, with each socket having its own dict.

Connectionless (UDP):-

① UDP socket identified by 2-tuple (Dest IP, Dest Port)

Flow control:- Transmission rate can be controlled by TCP, not by UDP

Overhead:- TCP header = 20 B, UDP header = 8 B

Chunk header into 16 bit segments, add them up (add carry back) and perform 1's complement to create a checksum

Receiver performs the same steps and compares against the checksum