Operating Systems and Networks

OS is the middleware b/w hardware and the application / user programs. It provides abstraction to the users for various syst 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.

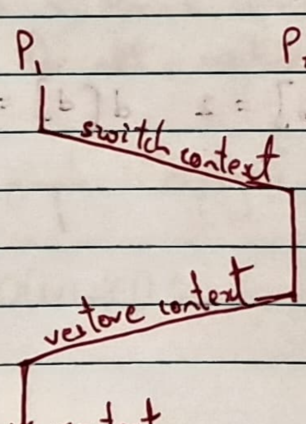
Process Virtualization :-

Process :- Running programs → set of instructions

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

Only running ~~process~~ programs consume resources.

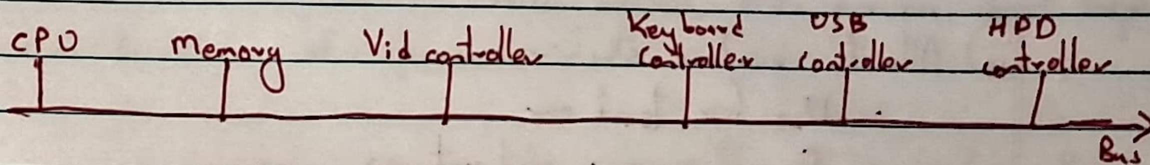
Each process has its own CPU. Multiple processes can run on the same CPU with the help of context switching (virtualization of the CPU)



apsara

Date: __.__.__

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 infor
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 retargs

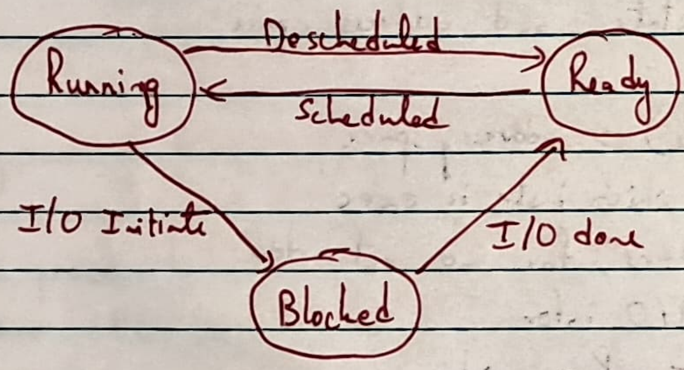
(iii) Creation of Prog heap:-
- Used for dynamic data like malloc and free

(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

apsara

Date: __.__.__

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 a process wait, suspend or find the status of a process.

APIs allow different programs to communicate with eachother. 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. This 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 ~~a~~ wait -> zombie process. Init adopts orphans and reaps them.

KV has doll-dole merger, but in [illegible] 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 ~~tech~~ level mechanism.

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

User Proc ⟶ Make syscall . . . . . . Ret from syscall

trap bit = 0 . . . . . . trap bit = 1

Execute sys call

Kernel Mode

Interrupt:- Signal that conveys that "something" 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 ~~also~~ called by the syscall.

Switching b/w Processes

(i) Cooperative [Non-preemptive 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 then 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, ~~send~~ 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. ~~the~~

What if during handling of one interrupt, another interrupt occurs?
Disable other interrupts!
Create sophisticated locking mechanism 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 ~~exe~~ run time. We just pick a process (rand or FCFS) and serialize.

· But if we relax the assumption that all proc take same time, FCFS is not a great idea.

We then choose to ~~sho~~ 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

Perf Metric:– Response Time

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