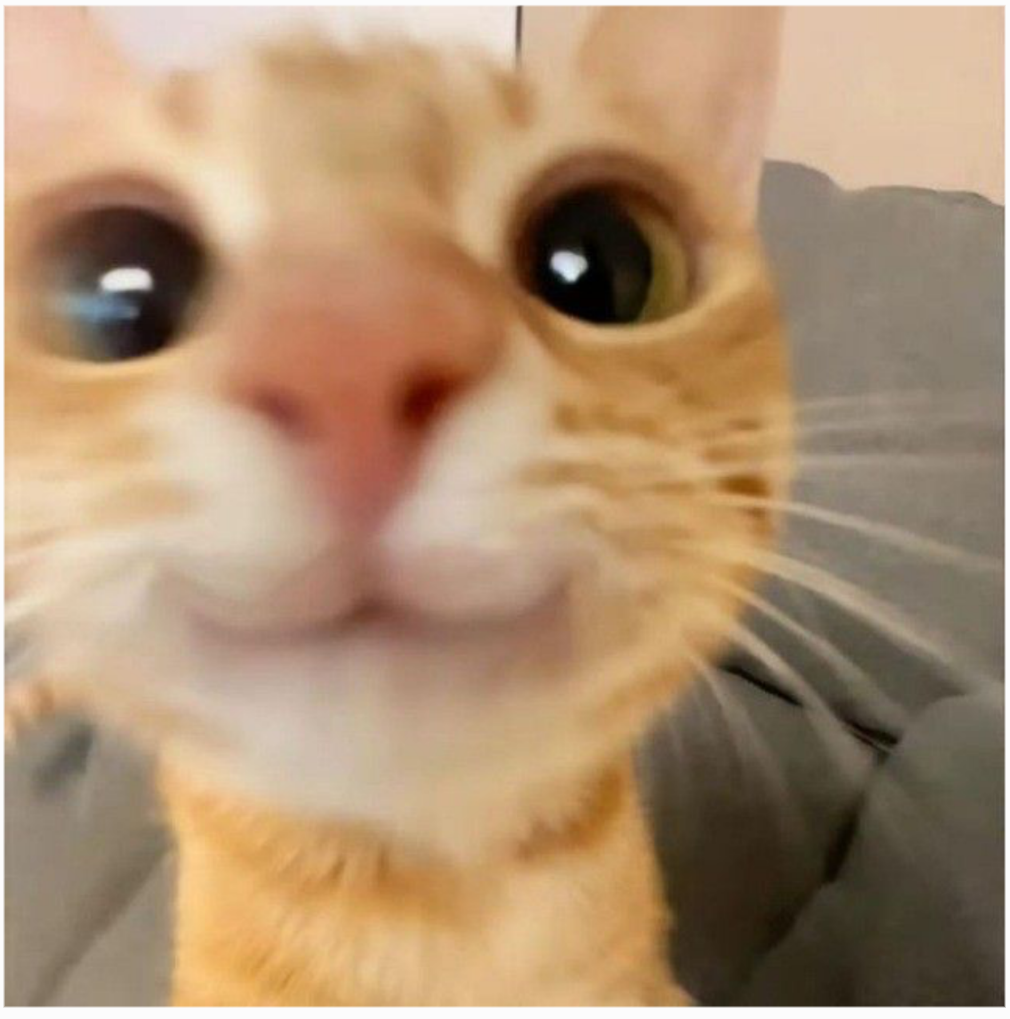
4. Direct Execution



To virtualize the CPU

the OS needs to share the physical CPU among many jobs running seemingly at the same time
=> Time Sharing

Challenges : performance / control

- performance : how can we implement virtualization without adding excessive overhead to the system?
- control : control the processes are mandatory to the OS (ex.resource management)

=> Obtaining high performance while maintaining control
=> Both hardware and OS support are required

Direct execution

: just run the program directly on the CPU => Fast

- Problems

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Figure 6.1: **Direct Execution Protocol (Without Limits)**

- How can the OS make sure the program doesn't do anything that we don't want it to do, while still running it efficiently?
- How does the OS stop process from running and switch to another process, thus implementing the time sharing?

=> Without limits on running programs, the OS wouldn't be in control of anything => thus would be "just a library"

Problem of Direct execution - #1 : Restricted Operations

What if the process wishes to perform some kind of **restricted operation**? (ex. issuing I/O request to a disk, gaining access to more system resources such as CPU or memory)

A process must be able to perform I/O and some other restricted operations, but HOW? => Limited Direct execution : limit what the process can do without OS assistance

=> Different modes of execution : user mode / kernel mode

- Trap instruction : jumps into the kernel / raises the privilege level to kerner mode
- return-from-trap instruction : returns into the calling user program / reducing the privilege level back to user mode

hardware must save enough of the caller's registers to return correctly when the OS issues the return-from-trap instruction.

Protect the OS during system calls

- By adding a hardware trapping mechanism
- ensuring all calls to the OS are routed through it

Trap table : stores the addresses of special function in the OS

- OS tells the hardware what code to run when certain exceptional events occur.
- The OS informs the hardware of the locations of trap handlers
- Kernel set up a trap table at boot time.
- OS can tell the hardware where the trap table resides in memory

System-call number

user code cannot specify an exact address to jump to
=> must request a particular service via number.
= This level of indirection serves as a form of protection

Limited Direct Execution

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of... syscall handler	

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call
		trap into OS
	save regs (to kernel stack)	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to PC after trap	
		...
		return from main
		trap (via `exit()`)
Free memory of process		
Remove from process list		

Figure 6.2: **Limited Direct Execution Protocol**

Problem of Direct execution - #2 : Switch between Processes

process is running on the CPU => the OS is not running
=> How the OS take an action if it is not running on the CPU?
=> How can the OS regain control of the CPU so that it can switch between processes?

A Cooperative Approach : Wait for System Calls

the OS trusts the processes to behave reasonably

1. Processes that run for too long are assumed to periodically give up the CPU (by making yield system call / transfer control to the OS)
=> so that the OS can decide to run some other task.

2. Applications also transfer control to the OS when they do something illegal

=> In a Cooperative scheduling system,
The OS regains control of the CPU by WAITING for a system call / illegal operation of some kind to take place.

But what if a process ends up in an infinite loop, and never makes a system call?
: Reboot => can run into this problem again.
without additional help from the hardware, the OS can't do much at all when a process refuses to make system calls.

The way the OS tries to handle malfeasance is to simply terminate the offender : One strike and you're out!

A Non-Cooperative Approach : The OS takes Control

Timer interrupt

- A timer device can raise an interrupt every so many milliseconds
- when the interrupt is raised
  - the currently running process is halted
  - pre-configured interrupt handler in the OS runs
- During the boot sequence, the OS must start the timer
- Once the timer has begun, the OS can feel safe in that control will eventually be returned to it

Saving and Restroring Context

decision has to be made : whether to continue running the currently-running process, or switch to a different one.
=> this made by the scheduler / part of the OS

If the decision is made to switch => OS executes a context switch

- context switch (a low-level piece of code)
- By switching stacks, the kernel enters the call to the switch code in the context of one process (the one that was interrupted) and returns in the context of another(the soon-to-be-executing)
- When the OS then finally executes a return-from-trap instruction, the soon-to-be-executing process becomes the currently-running process. => the context switch is complete.

Concurrency

What if another interrupt occurs, during interrupt or trap handling?
: OS might disable interrupts during interrupt processing
However, disabling interrupts for too long could lead to lost interrupts

=> locking : protect concurrent access to internal data structures.

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**