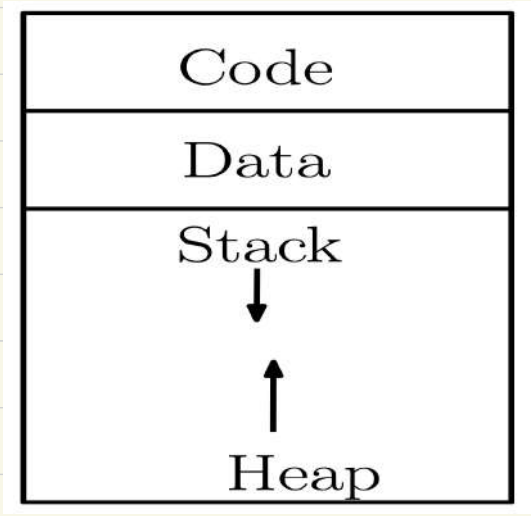
Limited Direct Execution

How to achieve virtualization of the CPU? → by time sharing.

The biggest challenge while virtualization
↳ How to run procesn efficiently while retaining control over the CPU?
i.e. High Performance while maintaining control.
↳ OS tries to do this with the help of hardware.

What is direct execution: → run the program directly in the CPU.



↓

* create entry for procesn list
* Allocate memory
* load program into memory
* Set up stack with argc/argv
* clear registers
* execute call main()

} things OS needs to do

Then the program → run main()
execute return from main

Finally, OS → free memory of process
Remove process from process list.

The above approach has two problems.

① How does the OS stops programs from running forever
(by time sharing)

② How does OS protect itself from buggy/malicious programs
(by restricted execution)

An analogy for restricted execution

The parent (OS) let the kid (program) cook.
- the kid is allowed in the kitchen
but can do only simple stuffs.

dangerous actions are not allowed.
- the kid can't use sharp knives, open the gas, etc.
- if they try, they must call the parent for help.

System calls → asking parents to do such things

Limited Direct execution = Balance of speed and safety.

- the kid does as much as possible on their own (faster)
  (lets user code run directly on hardware)
- for dangerous stuffs parents supervise (safe)

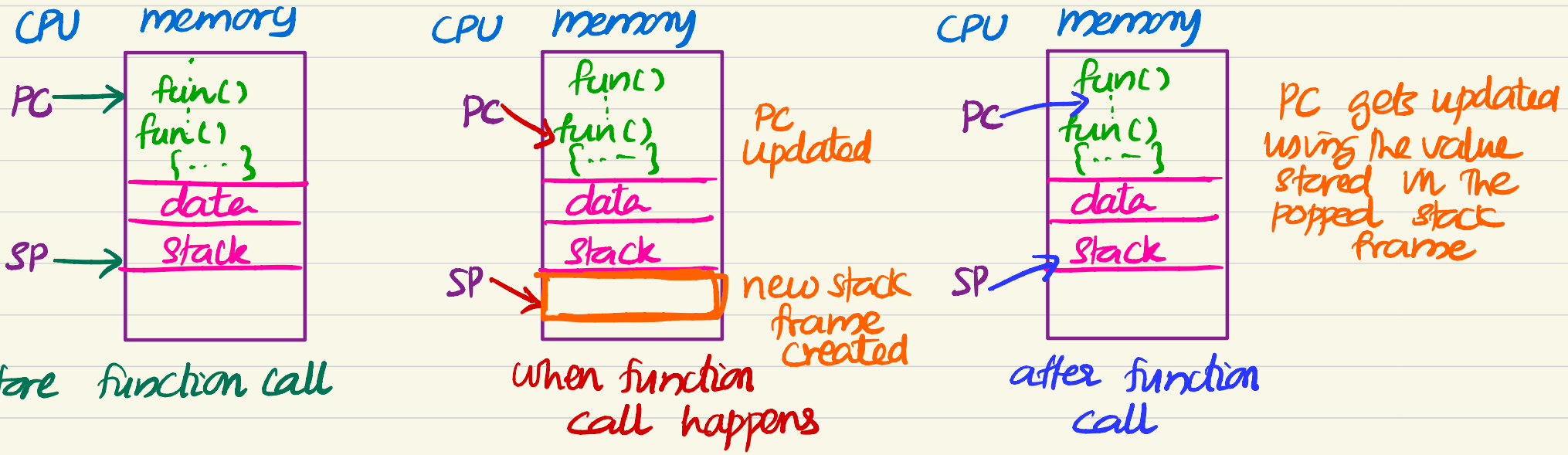
(controls transitions into priveleged operation using traps and system calls)

- parent doesn't hover constantly (only steps in when needed)

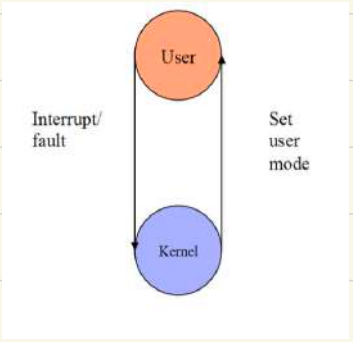
Before getting into a system call, let us see how a procedure call (or function call) works.

* A function call translates to a jump instruction
(it transfer control from the current location in the code to the location of the function)
* A new stack frame is pushed to stack and stack pointer (SP) gets updated.

* The new stack frame contains - return address (where to return after the function finishes)
  - arguments (the parameters passed to the function)
  - local variables (variables defined inside the function) etc..



What happens in system call?



* CPU hardware has multiple privilege levels → user, kernel

Kernel mode → to run OS code like system calls.

User mode → to run user code.

Kernel mode is more privileged, since certain instructions can only be executed in kernel mode.

* While switching b/w the two modes, kernel also need to store the context of the process, but it does not trust user stack.
  - it uses a separate kernel stack when in kernel mode.

* Also, kernel does not trust user provided address that indicates where to jump to

- Kernel sets up Interrupt Descriptor Table (IDT) or trap table at boot time. It contains the addresses of kernel functions for all system calls and other events that needs to

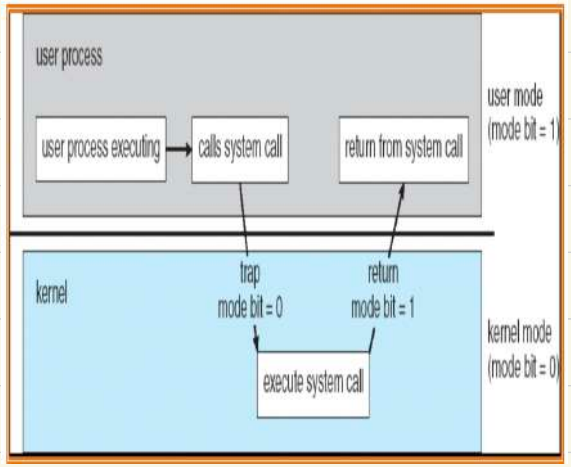
like, the address of the code to run when a hard disk interrupt takes place, when a keyboard interrupt occurs, or when a program makes a system call? etc.

Mechanism of a system call

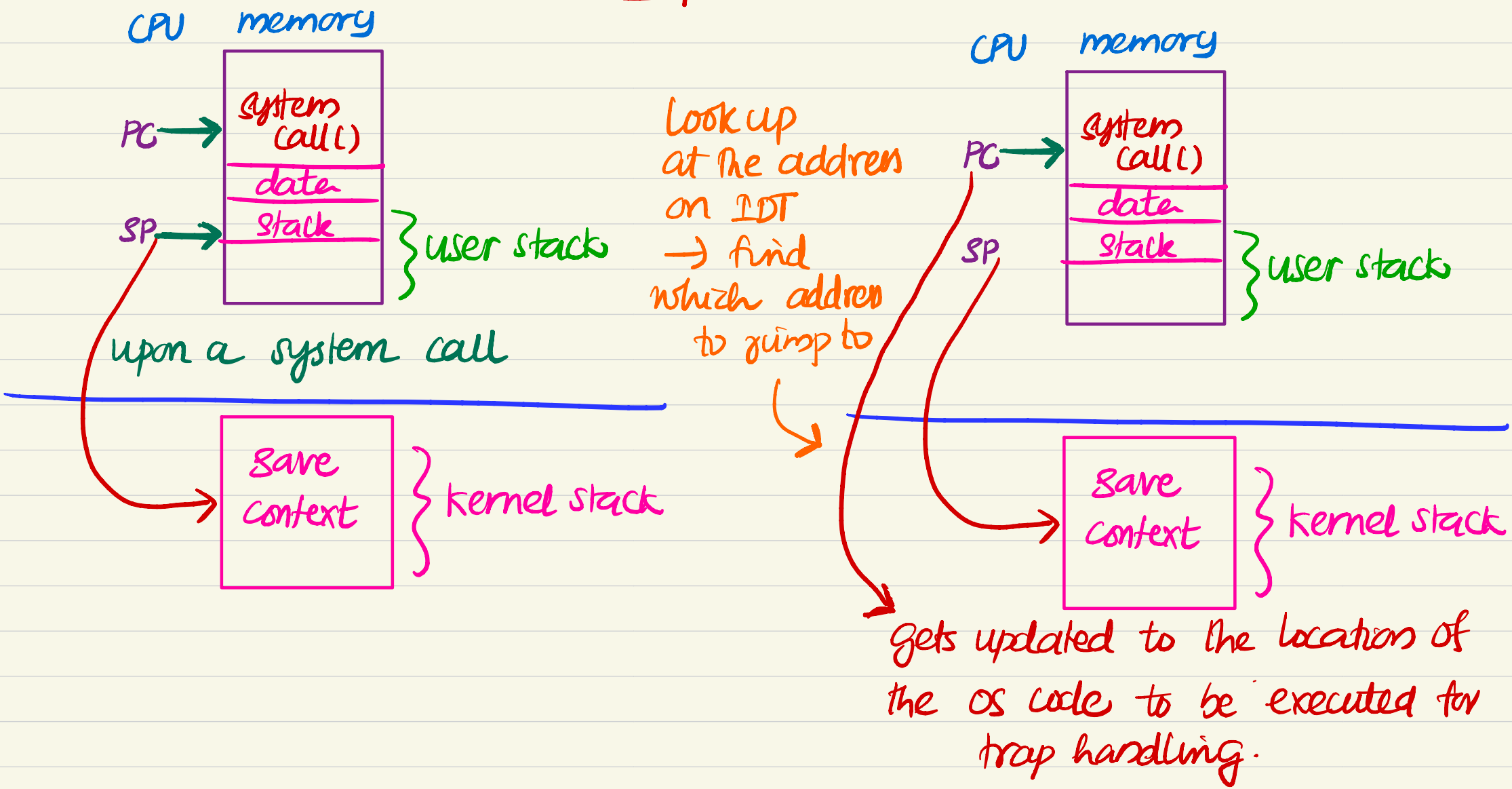
* When a system call is made → a special trap instruction CPU runs a

For instance, when you call functions like open() or read(), a hidden trap instruction inside the C library gets invoked.

* What does this trap instruction do?
  - save context (old PC, registers) to kernel stack
    OS maintains separate kernel stack for each process
  - it turns the CPU to higher privilege level (kernel mode)
  - look up address in IDT and jump to trap handler function in OS code



Trap


CPU
memory
PC
system call()
data
SP
stack
user stack
upon a system call
save context
kernel stack
Look up at the address on IDT → find which address to jump to
CPU
memory
PC
system call()
data
SP
stack
user stack
save context
kernel stack
gets updated to the location of the OS code to be executed for trap handling.

* when does the trap instruction gets executed on hardware (CPU)?
  - during a system call
  - program fault (program does something illegal)
    like, divide by zero, invalid memory access, page fault (access a valid virtual addres not currently in RAM), etc..
  - interrupt happens ; like keyboard key pressed, disk read finished etc..

In all the scenarios, we follow the same mechanism explained earlier.

* How does OS know whether the user issued a valid syscall or not?

For this, every system call has been assigned a number, a system-call number. The user code place the desired system call number in a register so that
before OS executes the trap handler function, it first examines

this number, ensures it is valid, and if it is, executes the corresponding code. This extra protection via number is needed (think, why?).

How to return from the trap?

When OS finishes handling the system call, it calls a special instruction return from trap.

How it is been done? – CPU restore registers from kernel stack
– move to user mode
– jump to PC after trap

Now, the user program continues the execution, as if nothing has happened.

Summary of limited direct execution protocol

OS @ boot (kernel mode)	Hardware	Program (user mode)
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**

How to switch b/w processes?

Before answering how, why would OS switch b/w processes?

- for time sharing
- Handling a blocking system call for the current process (eg:- waiting for disk I/O, network data, user input etc.)
- process has terminated (eg:- due to some segmentation fault)

On the above scenarios, OS performs a context switch to switch from one process to another. Now, we will get to see, how OS switches b/w processes?

① A co-operative approach: wait for system calls

* On this style, OS trusts the processes, and believes that they periodically give up the CPU so that the OS can decide to run some other task.

* a friendly process would yield system call, so that OS gets back the control and can run other processes.

* Processes also pass the control when they do something illegal.

eg:- early versions of Macintosh used to follow this approach.

problem with this approach: a malicious process can ends up in an infinite loop and never makes a system call.

↳ only way for OS to get back the control is "reboot the machine".

(2) A non-co-operative approach: the OS takes the control

How does OS gain the control of the CPU even if the proces are not being co-operative? a timer interrupt

* during the boot sequence, the OS start the timer.
* a timer device is programmed to raise an interrupt every milliseconds.
* when the interrupt is raised
  - the current process is halted
  - save the state of the program
  - a pre-configured interrupt handler in the OS runs.

will learn in the next chapter

Once the interrupt happens → scheduler decides whether to switch or not.

If the decision is to switch → context switch happens.
a low-level piece of code executed by OS.

When a timer interrupt happens
as before, CPU save registers (current process) → K-stack (current process),
move to kernel mode, and jump to trap handler.

Now, OS while handling the trap, if it decides to switch the process,
call a switch() routine.
process control block
↳ save regs (current process) → PCB (A)
restore regs (next process) ← PCB (B)
switch to K-stack (next process)
return from trap (into next process)

Again CPU, restore regs (next pros) ← K-stack (next process)
move to user mode
jump to the PC of this process — new process resumes

Example of a context switch

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)**

Reference: Chapter-6 of OSTEP

Remark: what happens when you are handling one interrupt and another one happens, or two interrupts happen together

↳ will deal these problems in the "second piece of the book, concurrency!