6 Mechanism: Limited Direct Exec.

* OS must share the physical CPU among many jobs in order to virtualize it.
  ↳ time-sharing : run one process on CPU for a bit then switch to another.

* challenges :
  1. performance : desire to implement virtualization w/o adding too much overhead.
  2. control : how to run processes efficiently w/ maintaining control over CPU
     ↳ lose control ⇒ process hog CPU

* require both hardware & OS support to address these.

6.1 LIMITED DIRECT EXECUTION

* Limited direct execution : technique used by OS devs to address these issues ↑
  ↳ run the program directly on the CPU. (while limiting what it can do).

  * OS creates a process entry for the program it wants to run in a process list
  * " allocates memory for it
  * " loads code into memory from disk
  * " locates entry point & jumps to it
  * " start executing code.
  * after execution, free memory of process
  * " " , remove from process list.

7 hurdles that are left unaddressed by this procedure.

1. How does OS ensure program won't do anything we don't want it to do? (while still running efficiently)
2. How does OS perform context switch?

(6.2) Problem 1: Restricted Operations

* a running process may want to do some restricted operations:
    * I/O request to disk
    * gain access to more system resources (CPU/RAM)

CRUX: Process must be able to perform I/O & other restricted operations w/o OS giving complete control over to it.

* how does system differentiate between system calls & procedure calls?
    - ↳ syscalls are procedure calls w/ a trap instruction.
    - ↳ library places syscall ID & arguments into special registers / stack locations then executes a trap instruction. (into kernel)
    - ↳ library unpacks return values after trap & returns control from kernel to program.

* If we let any process issue I/O w/o restricting it then file permissions are pointless...

* Processor Modalities: (CPU modes)
    1. User mode: code that runs in user-mode is restricted. (process can't execute I/O)
        - ↳ if process tries restricted operations while running in user-mode
        - ⟹ processor raises exception & OS kills process.
    2. Kernel mode: privileged mode which the OS runs in.
        - ↳ code that runs in kernel mode may execute restricted operations.

has full access to system resources

User-Mode
* wishes to do restricted operation
* syscall

doesn't have full access to hardware

Kernel Mode
* safe code that has access to restricted operation

* trap instruction
* return from trap

* syscalls allow kernel to expose certain key functionality to user programs (e.g.)
    * accessing FS
    * create/destroy processes
    * communicate to other processes
    * allocating more memory.

* trap instructions : jump into kernel & raise privilege level
* return from trap instructions : return to calling program & reduces privilege level

* hardware must save enough of caller's registers to return from trap correctly.
  ↳ x86 processor pushes caller's registers onto kernel stack & pops off when returning.

* Problem : how does trap know what code to execute inside the OS?
  ↳ what if it jumps to the wrong place & executes the wrong code?

    * DON'T let calling process choose address in kernel to jump to.
    * Instead, kernel sets up trap table @ boot time (in kernel-mode)
        ↳ locations of trap handlers assigned to specific traps/exceptions. (trap = trap handler addr.)
        ↳ this way, hardware knows where to jump when a trap is called.
            (until reboot)

OS (kernel) | HARDWARE | PROGRAM (user)

boot

#	OS (kernel)	Hardware	Program (user)
1	initialize trap table		
2		store locations of trap handlers	
3	initialize user process create entry for process list		
4	allocate mem for program		
5	load program into mem		
6	setup user stack		
7	fill kernel stack w/ reg/PC		
8	return from trap		
9		restore regs from kernel stack to user stack	
10		move to user mode	
11		jump to entry point	
12	run process		start executing ⋮
13	handle privileged operation		call syscall
14			trap into OS
15		save regs to kernel stack	
16		move to kernel mode	
17		jump to trap handler	
18	handle trap		
19	do work of syscall		
20	return from trap		
21	continue process	restore regs from kernel stack	
22		to user stack move to user mode	
23		jump to PC	⋮
24			return from program
25	clean up process		trap (exit)
26	free mem of process		
27	remove from process list		

TIP: Ensure user passes valid args to syscall.

* user program is required to specify correct syscall ID in register before trap

6.3 Problem 2: Switching Between Processes

* **Issue:** consider a single CPU machine, then if one process is running that must mean the OS is currently not running. How does OS switch between processes?

CRUX: How to regain control of the CPU?

* **A Cooperative Approach: Wait For Syscalls**
  * OS trusts the processes of the system to behave responsibly (first mistake!)
    ↳ assumes that a process will periodically give up CPU if running for too long
    ↳ most processes end up transferring control to OS frequently w/ system calls.
  * These types of systems also typically include a yield syscall for process to intentionally give up CPU usage w/o needing to do a restricted operation.
  * Generally, if a program does something illegal (access restricted memory), it will generate a trap into OS.

  * Q: what if a non-malicious process gets caught in infinite loop?
    ↳ how does OS regain control? (A: REBOOT!)

* **Non-Cooperative Approach: OS Takes Control**
  * OS really can't do much if a process doesn't make syscalls.

CRUX: How to regain control w/o cooperation?

* A: implement a timer interrupt, i.e.,
  * for every specific interval of time, raise an interrupt
  * OS implements pre-configured interrupt handler and kernel starts running.
  * OS can now kill, switch, or do whatever it pleases w/ the process.

@ Boot Time, tells hardware where timer handler is & starts timer.

NOTE: Timer can be turned off but it is a privileged operation.

* Hardware must save enough state of a process (PC, SP, params, etc.) when timer interrupt is triggered so it can start running again when return-from-trap back into the same process eventually. (could push onto kernel stack as w/ syscalls).

* Saving & Restoring Context:
  * OS, after regaining control (through syscall / timer interrupt), decides to either continue running same process or some other one. (scheduler)

* Context Switch:

through ass. code

① Save enough state (registers) of previously running process (so it can run again)
② Load enough state to " " soon-to-be " "

done by kernel.

Ensures that after return-from-trap instruction is executed, the system will resume execution of the new process instead of the old one.

	OS (kernel)	HARDWARE	PROGRAM (user)
	@ boot time		
1	initialize trap table		
2		save locations of syscall handlers	
3		" " " timer handler	
4	start interrupt timer		
5		Start timer	
			⋮
6			Process A is running
7		timer interrupt	
8		save regs(A) → k-stack(A) (saved in hardware registers for kernel (CPU))	
9		move to kernel mode	
10		jump to timer trap handler	
11	handle trap		
12	call switch() routine:	hardware ⟷ stored in memory (RAM)	
13	save regs(A) → proc_t(A)		
14	load regs(B) ← proc_t(B)		
15	switch to k-stack(B)] changing SP (CPU)		
16	return from trap (into B)		
17		load regs(B) ← k-stack(B)	
18		move to user mode	
19		jump to B's PC	
20			Process B is running

NOTE: Each process (& thread in Linux) has its own kernel-stack, used when a process traps into the kernel, allowing for execution in kernel mode

6.4 Worried Ab Concurrency?

Q: what happens when, during a syscall, a timer interrupt occurs?

i.e. interrupt occurs when already handling one?

- OS might disable interrupts when processing an interrupt.
  - ↳ can lose some interrupts this way.
- OS might have locking schemes to protect concurrent access to internal data structures.

↳ More detail later

6.5 Summary

- limited direct execution:
  1. setup hardware to limit what a process can do
  2. Run whatever program you want

— Analogous to BABY PROOFING

NOTE: syscalls & context-switches are time expensive.