- run one process for a little while, then run another one, and so forth. By sharing the CPU in this manner, virtualization is achieved.
- There are a few challenges in building such virtualization machinery. The first is performance: how can we implement virtualization without adding excessive overhead to the system?
- How can we run processes efficiently while retaining control over the CPU?
    - without control, a process could simply run forever and take over the machine, or access information that it should not be allowed to access. Obtaining high performance while maintaining control is thus one of the central challenges in building an operating system.

Basic Technique: Limited Direct Execution

- To make a program run as fast as one might expect, OS developers came up with a technique, which we call limited direct execution.
- The "direct execution" part of the idea is simple: just run the program directly on the CPU.
    - Thus, when the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory (from disk), locates its entry point (i.e., the main() routine or something similar), jumps to it, and starts running the user's code

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)**

- this approach gives rise to a few problems in our quest to virtualize the CPU.
    - if we just run a program, 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?
    - when we are running a process, how does the operating system stop it from running and switch to another process, thus implementing the time sharing we require to virtualize the CPU?

Problem #1: Restricted Operations

- A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system. How can the OS and hardware work together to do so?
    - the approach we take is to introduce a new processor mode, known as user mode; code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issue I/O requests; doing so would result in the processor raising an exception; the OS would then likely kill the process.

    - In contrast to user mode is kernel mode, which the operating system (or kernel) runs in. In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions
- The hardware assists the OS by providing different modes of execution. In user mode, applications do not have full access to hardware resources. In kernel mode, the OS has access to the full resources of the machine.
    - Special instructions to trap into the kernel and return-from-trap back to user-mode programs are also provided, as well as instructions that allow the OS to tell the hardware where the trap table resides in memory.
    - The trap is a signal raised by a user program instructing the operating system to perform some functionality immediately
    - A trap table is what is conventionally used by the system call handler to invoke the requested operating service routine
- What should a user process do when it wishes to perform some kind of privileged operation, such as reading from disk?
    - To enable this, virtually all modern hardware provides the ability for user programs to perform a **system call.**
    - system calls allow the kernel to carefully expose certain key pieces of functionality to user programs, such as accessing the file system, creating and destroying processes, communicating with other processes, and allocating more memory.
    - To execute a system call, a program must execute a special trap instruction. This instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode;
        - Once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), and thus do the required work for the calling process.
        - When finished, the OS calls a special return-from-trap instruction, which returns into the calling user program while simultaneously reducing the privilege level back to user mode.
- The hardware needs to be a bit careful when executing a trap
    - it must make sure to save enough of the caller's registers in order to be able to return correctly when the OS issues the return-from-trap instruction.
    - On x86, for example, the processor will push the program counter, flags, and a few other registers onto a per-process kernel stack; the return-from trap will pop these values off the stack and resume execution of the usermode program
- how does the trap know which code to run inside the OS?
    - The kernel does so by setting up a trap table at boot time. When the machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as needed.
    - One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur. For example, what code should run when

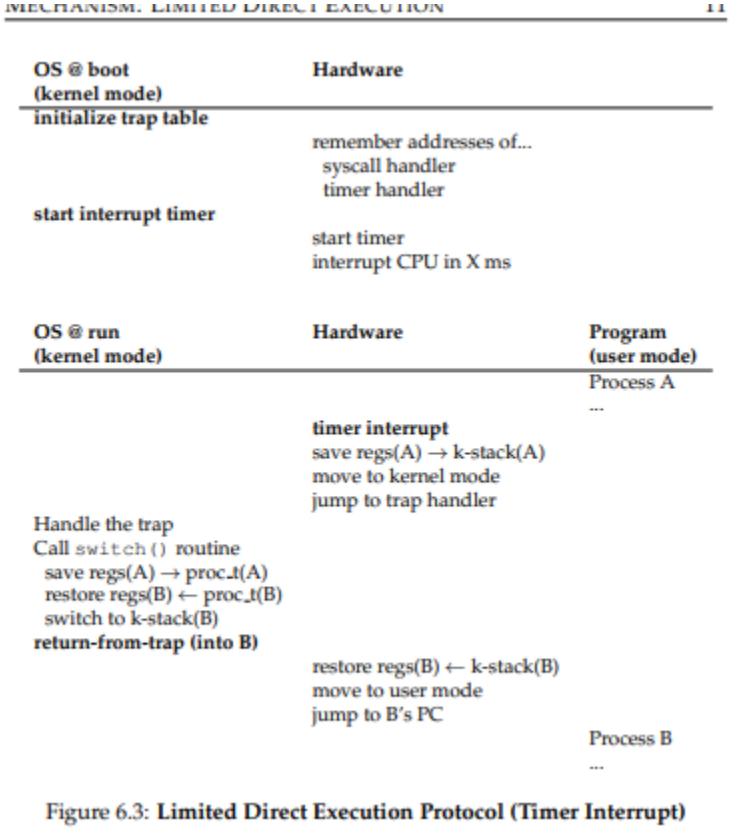
a harddisk interrupt takes place, when a keyboard interrupt occurs, or when a program makes a system call?
        - The OS informs the hardware of the locations of these trap handlers, usually with some kind of special instruction. Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do (i.e., what code to jump to) when system calls and other exceptional events take place.
- To specify the exact system call, a system-call number is usually assigned to each system call. The OS, when handling the system call inside the trap handler, examines this number, ensures it is valid, and, if it is, executes the corresponding code. This level of indirection serves as a form of protection;
- Being able to execute the instruction to tell the hardware where the trap tables are is a very powerful capability. Thus it is also a privileged operation. If you try to execute it in user mode, the hardware won't let you
- There are two phases in the limited direct execution (LDE) protocol. In the first (at boot time), the kernel initializes the trap table, and the CPU remembers its location for subsequent use. The kernel does so via privileged instruction.
    - when running a process, the kernel sets up a few things (e.g., allocating a node on the process list, allocating memory) before using a return-from-trap instruction to start the execution of the process; this switches the CPU to user mode and begins running the process.
    - When the process wishes to issue a system call, it traps back into the OS, which handles it and once again returns control via a return-from-trap to the process.
    - The process then completes its work, and returns from main(); this usually will return into some stub code which will properly exit the program (say, by calling the exit() system call, which traps into the OS).
    - At this point, the OS cleans up and we are done.

Problem #2: Switching Between Processes

- The next problem with direct execution is achieving a switch between processes.
- If a process is running on the CPU, this by definition means the OS is not running.
- There is clearly no way for the OS to take an action if it is not running on the CPU.
- How can the operating system regain control of the CPU so that it can switch between processes?
    - One approach that some systems have taken in the past is known as the cooperative approach. In this style, the OS trusts the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.
    - How does a friendly process give up the CPU?
        - Most processes transfer control of the CPU to the OS quite frequently by making system calls, for example, to open a file and subsequently read it, or to send a message to another machine, or to create a new process.
        - Systems like this often include an explicit yield system call, which does nothing except to transfer control to the OS so it can run other processes

    - Applications also transfer control to the OS when they do something illegal. For example, if an application divides by zero, or tries to access memory that it shouldn't be able to access, it will generate a trap to the OS.
        - The OS will then have control of the CPU again (and likely terminate the offending process).
    - Thus, in a cooperative scheduling system, the OS regains control of the CPU by waiting for a system call or an illegal operation of some kind to take place.
- How can the OS gain control of the CPU even if processes are not being cooperative? What can the OS do to ensure a rogue process does not take over the machine?
    - Without some additional help from the hardware, it turns out the OS can't do much at all when a process refuses to make system calls
    - In the cooperative approach, your only recourse when a process gets stuck in an infinite loop is to reboot the machine.
    - The solution: a timer interrupt.
        - A timer device can be programmed to raise an interrupt every so many milliseconds; when the interrupt is raised, the currently running process is halted, and a pre-configured interrupt handler in the OS runs.
        - At this point, the OS has regained control of the CPU, and thus can do what it pleases: stop the current process, and start a different one
        - the OS must inform the hardware of which code to run when the timer interrupt occurs; thus, at boot time, the OS does exactly that.
        - Second, also during the boot sequence, the OS must start the timer, which is of course a privileged operation. Once the timer has begun, the OS can thus feel safe in that control will eventually be returned to it, and thus the OS is free to run user programs. The timer can also be turned off
- Now that the OS has regained control, whether cooperatively via a system call, or more forcefully via a timer interrupt, a decision has to be made: whether to continue running the currently-running process, or switch to a different one. This decision is made by a part of the operating system known as the scheduler;
- If the decision is made to switch, the OS then executes a low-level piece of code which we refer to as a context switch.
    - A context switch is conceptually simple: all the OS has to do is save a few register values for the currently-executing process (onto its kernel stack, for example) and restore a few for the soon-to-be-executing process (from its kernel stack).
    - By doing so, the OS thus ensures that when the return-from-trap instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.
    - To save the context of the currently-running process, the OS will execute some low-level assembly code to save the general purpose registers, PC, and the kernel stack pointer of the currently-running process, and then restore said registers, PC, and switch to the kernel stack for the soon-to-be-executing process. 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 one)
    - When the OS then finally executes a return-from-trap instruction, the soon-to-be-executing process becomes the currently-running process. And thus the context switch is complete.
- The addition of a timer interrupt gives the OS the ability to run again on a CPU even if processes act in a non-cooperative fashion. Thus, this hardware feature is essential in helping the OS maintain control of the machine
- In the example:
    - Process A is running and then is interrupted by the timer interrupt. The hardware saves its registers (onto its kernel stack) and enters the kernel (switching to kernel mode).
    - In the timer interrupt handler, the OS decides to switch from running Process A to Process B.
    - At that point, it calls the switch() routine, which carefully saves current register values (into the process structure of A), restores the registers of Process B (from its process structure entry), and then switches contexts, specifically by changing the stack pointer to use B's kernel stack (and not A's).



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)**

    - Finally, the OS returns from-trap, which restores B's registers and starts running it
- there are two types of register saves/restores that happen during this protocol. The first is when the timer interrupt occurs; in this case, the user registers of the running process are implicitly saved by the hardware, using the kernel stack of that process.
- The second is when the OS decides to switch from A to B; in this case, the kernel registers are explicitly saved by the software (i.e., the OS), but this time into memory in the process structure of the process.

- what happens when, during a system call, a timer interrupt occurs? or What happens when you're handling one interrupt and another one happens? Doesn't that get hard to handle in the kernel?

    - the OS does indeed need to be concerned as to what happens if, during interrupt or trap handling, another interrupt occurs
    - This is the topic of concurrency.
    - One simple thing an OS might do is disable interrupts during interrupt processing; doing so ensures that when one interrupt is being handled, no other one will be delivered to the CPU
    - Operating systems also have developed a number of sophisticated locking schemes to protect concurrent access to internal data structures.

Summary:

- limited direct execution: just run the program you want to run on the CPU, but first make sure to set up the hardware so as to limit what the process can do without OS assistance.
- the OS “baby proofs” the CPU, by first (during boot time) setting up the trap handlers and starting an interrupt timer, and then by only running processes in a restricted mode. By doing so, the OS can feel quite assured that processes can run efficiently, only requiring OS intervention to perform privileged operations or when they have monopolized the CPU for too long and thus need to be switched out.