LECTURE 7: LIMITED DIRECT EXECUTION

OSTEP:Chapter 6

Objectives

- Execution of a Process
- How does a user process access restricted operations?
- Virtualization of Multiprocessor: Timesharing

What is a Process?

execution stream

Process: An execution stream in the context of a process state

What is an execution stream?

- Stream of executing instructions
- Running piece of code
- "thread of control"

What is process state?

- Everything that the running code can affect or be affected by
- Registers
  - General purpose, floating point, status, program counter, stack pointer
- Address space
  - Heap, stack, and code
- Open files

Processes vs. Programs

A process is different than a program

- Program: Static code and static data
- Process: Dynamic instance of code and data

Can have multiple process instances of same program

- Can have multiple processes of the same program
  Example: many users can run “ls” at the same time

How to Provide Good CPU Performance?

Direct execution

- Allow user process to run directly on hardware
- OS creates process and transfers control to starting point (i.e., main())

Problems with direct execution?

1. Process could do something restricted
   Could read/write other process data (disk or memory)
2. Process could run forever (slow, buggy, or malicious)
   OS needs to be able to switch between processes
3. Process could do something slow (like I/O)
   OS wants to use resources efficiently and switch CPU to other process

Solution:

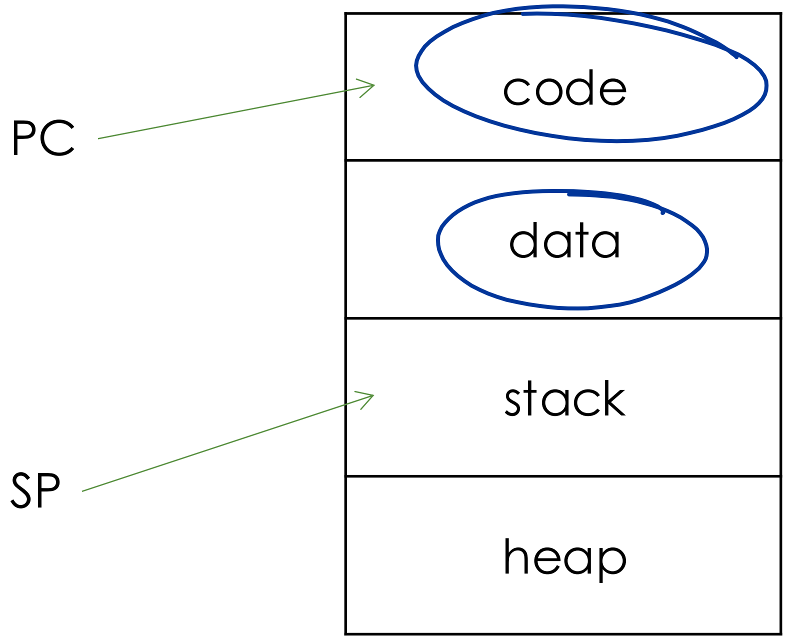
Limited direct execution – OS and hardware maintain some control

Execution of a process

register will store values as well.

- OS allocates memory and creates memory image
  - Code and data (from exe)
  - Stack and heap
- Points CPU program counter to current instruction
  - Other registers may store operands, return values etc.
- After setup, OS is out of the way and process executes directly on CPU

Program counter to current instruction.



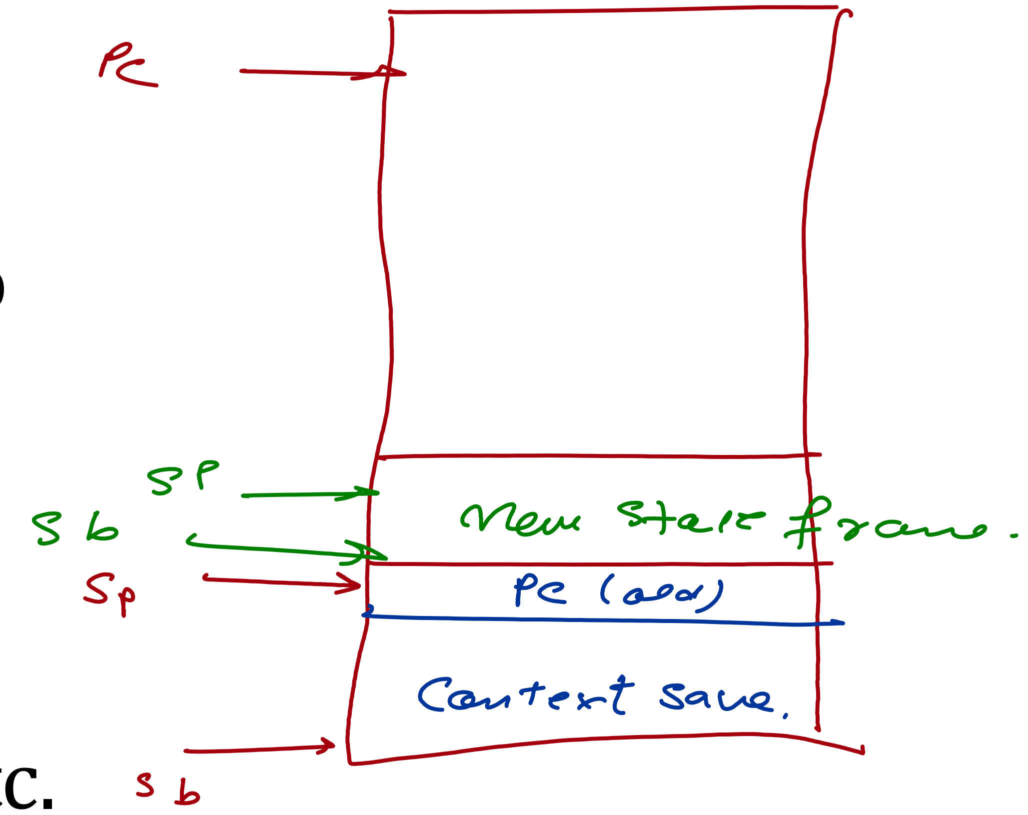
.exe → memory image.

(Jump instruction)

Caller → callee

What happens during Function Call?

- caller to callee – via jump instruction
- New Stack frame created
- Old PC (ret value) pushed to stack
- New PC updated to callee
- Stack frame contains return value, function arguments etc.



System Call vs Function Call

- CPU hardware has multiple privilege levels – One to run user code: user mode
  - One to run OS code like system calls: kernel mode
  - Some instructions execute only in kernel mode
- Kernel does not trust user stack
  - Uses a separate kernel stack when in kernel mode
- Kernel does not trust user provided addresses to jump to
  - Kernel sets up Interrupt Descriptor Table (IDT) at boot time
  - IDT has addresses of kernel functions to run for system calls and other events

Mechanism of system call: trap instruction

- When system call must be made, a special trap instruction is run (usually hidden from user by libc)
- Trap instruction execution
  - Move CPU to higher privilege level
  - Switch to kernel stack
  - Save context (old PC, registers) on kernel stack
  - Look up address in IDT and jump to trap handler function in OS code

CPU * (some bit to mark the privilage level)

userlevel

trap

kernel

kerne stack

context save.

Switch process

A → B

A → kerne (A) → kernel (B) → B

More on the trap instruction

(i) sys-call
(ii) interrupt
(iii) Program fault.

- Trap instruction is executed on hardware in following cases:
  - System call (program needs OS service)
  - Program fault (program does something illegal, e.g., access memory it doesn't have access to)
  - Interrupt (external device needs attention of OS, e.g., a network packet has arrived on network card)
- Across all cases, the mechanism is: save context on kernel stack and switch to OS address in IDT
- IDT has many entries: which to use?
  - System calls/interrupts store a number in a CPU register before calling trap, to identify which IDT entry to use

Return from TRAP

special instruction (return-from-trap)

- When OS is done handling syscall or interrupt, it calls a special instruction return-from-trap
  - Restore context of CPU registers from kernel stack
  - Change CPU privilege from kernel mode to user mode
  - Restore PC and jump to user code after trap

restore context from kernel stack → change Previlage LVL on CPU →

- User process unaware that it was suspended, resumes execution as always
- Must you always return to the same user process from kernel mode? No (we can go to other process also).
- Before returning to user mode, OS checks if it must switch to another process

Why switch between processes?

- Sometimes when OS is in kernel mode, it cannot return back to the same process it left
  - Process has exited or must be terminated (e.g., segfault)
  - Process has made a blocking system call
- Sometimes, the OS does not want to return back to the same process
  - The process has run for too long
  - Must timeshare CPU with other processes
- In such cases, OS performs a context switch to switch from one process to another

A → kerne (A) → kerne (B) → B

⇓ restoring context.

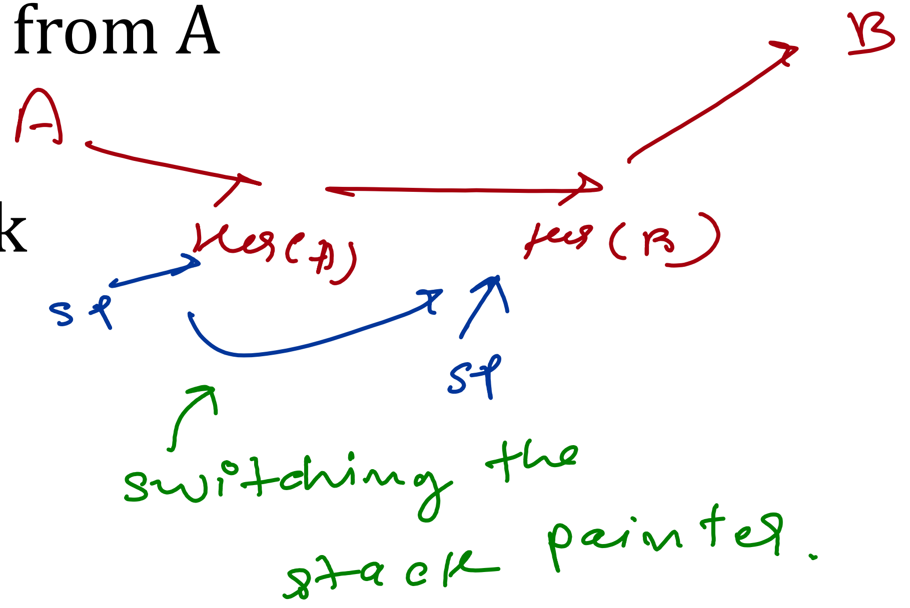
The OS scheduler

(i) Policy to pick
(ii) Switch to a process.

- OS scheduler has two parts
  - Policy to pick which process to run
  - Mechanism to switch to that process (this lecture)
- ~~Non preemptive~~ (cooperative) schedulers are polite
  - Switch only if process blocked or terminated
- Preemptive (non-cooperative) schedulers can switch even when process is ready to continue
  - CPU generates periodic timer interrupt
  - After servicing interrupt, OS checks if the current process has run for too long

Mechanism of context switch

- Example: process A has moved from user to kernel mode, OS decides it must switch from A to B
- Save context (PC, registers, kernel stack pointer) of A on kernel stack
- Switch SP to kernel stack of B
- Restore context from B's kernel stack
- Who has saved registers on B's kernel stack?
  - OS did, when it switched out B in the past
- Now, CPU is running B in kernel mode, return-from-trap to switch to user mode of B



A subtlety on saving context

context saved on kernel stack → (i) user mode → kernel mode. (ii) context switching.

- Context (PC and other CPU registers) saved on the kernel stack in two different scenarios
- When going from user mode to kernel mode, user context (e.g., which instruction of user code you stopped at) is saved on kernel stack by the trap instruction
  - Restored by return-from-trap
- During a context switch, kernel context (e.g., where you stopped in the OS code) of process A is saved on the kernel stack of A by the context switching code
  - Restores kernel context of process B

Thank You