Limited Direct Execution

Chapter 6

Preface: Function Calls

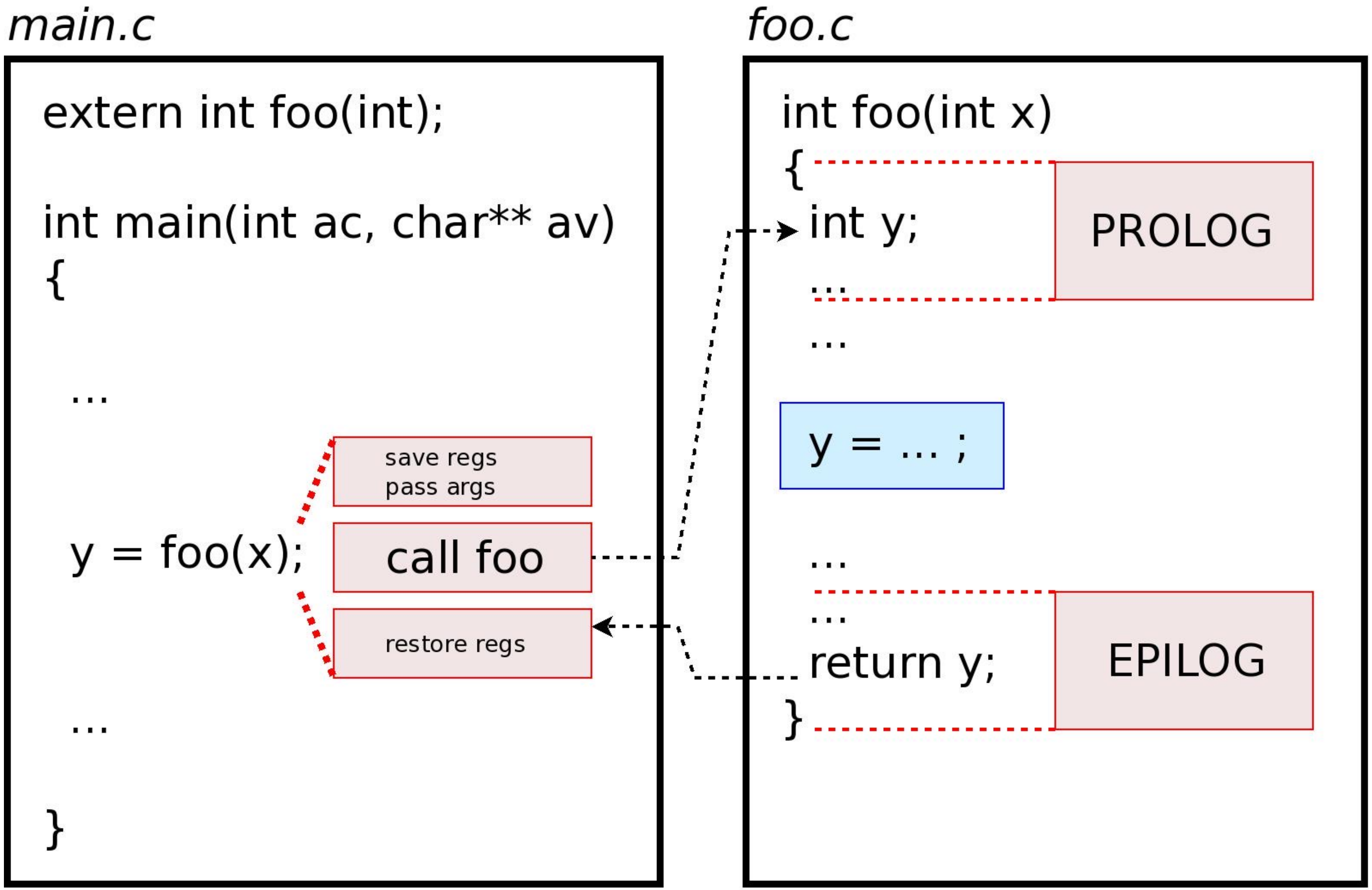

Image from: https://www.embeddedrelated.com/showarticle/172.php

CPU Virtualization Mechanism

- To share the CPU, we need a way to:
  - Run a process on the CPU
  - Provide security for sensitive operations
  - Switching between jobs
- Need a way to do this efficiently and maintain control over the system
  - Requires both **hardware** and **operating-system support**

Direct Execution

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	

- Any issues with this?
  - We cannot swap a process out for another one unless it gives control back to the OS and no support for privileged functionality

Operation Permissions

- Provide operation modes for the processor
  - User mode – basic operations that require minimal privileges
  - Kernel mode – full permission to all operations/resources provided by the OS (the OS is also referred to as the **kernel**, thus the name)

- Attempting to run privileged instructions in user mode will cause and exception

- We expose **system calls** to user mode so a request for the privileged functionality can be performed by the OS

Executing System Calls: User -> Kernel

- At boot, the **trap table** is setup in hardware to initialized all the functions to handle the system calls
- As part of a system call, a special instruction called a trap is executed
  - User mode code only know what system call is needed, but **NOT** where the system call code is located (Why?)
- The trap tells the hardware to:
  - save the state/context of the current process to a kernel stack (we'll need to resume later)
  - switch permission to kernel mode
  - load up the appropriate code to handle the trap for the OS

Executing System Calls: Kernel -> User

- When the OS is done running the code to handle the system call it executes a **return-from-trap**

- The return-from-trap tells the hardware to:
  - Restore the state/context for the program that called the trap
  - Switch back to user mode for instruction execution
  - Resume the program after the trap using the Program Counter (PC)

Limited Direct Execution (LDE)

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		

CPU Virtualization Mechanism

- To share the CPU, we need a way to:
  - ~~Run a process on the CPU~~
  - ~~Provide security for sensitive operations~~
  - **Switching between jobs (controlling process execution)**
- Need a way to do this efficiently and maintain control over the system
  - Requires both **hardware** and **operating-system support**

Cooperative Approach

- The OS expects that all programs will behave correctly and respect sharing of system resources
- Control is only transferred to the OS for system calls, illegal operations (perhaps an error), a **yield** call to simply allow for another process to take precedence
- Any issues with this?
  - Not a perfect word, relies on the developer to make the program share, no bugs like infinite loop.

Preemptive Approach (non-cooperative)

- A simple solution is to provide a timer device in hardware
- The timer is started during the OS boot process
- Each time a certain duration of time elapses (X milliseconds perhaps) a **timer interrupt** occurs
- This interrupt causes a trap that returns control back to the OS

Context Switching

- The OS may not switch back to the same process
  - a process has exited or must be terminated (e.g., segfault
  - a process has made a blocking system call
  - a timer interrupt occurs to give the CPU to another process (determined by the **scheduler**)

- The OS executes **context switch** code to swap the two processes

- Context switch code saves state/context from the current process and exchanges those values for a different ready process

Saving Context

- When moving from user to kernel mode process state/context is saved to the kernel stack by the hardware during the trap instruction
  - This is restored via return-from-trap

- During a context switch the hardware still saves process state/context to the kernel stack but the OS also:
  - Explicitly saves the state/context to the process table entry of the previously running process and restores the state/context of a ready process
  - Switches to the kernel stack of the ready process
  - Returns from trap using the ready process

LDE Timer Interrupt 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) $\rightarrow$ k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) $\rightarrow$ proc_t(A) restore regs(B) $\leftarrow$ proc_t(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) $\leftarrow$ k-stack(B) move to user mode jump to B's PC	
		Process B ...