
Chap.6

Limited Direct Execution

Problems of Direct Execution

- The OS must virtualize the CPU in an **efficient** manner while retaining **control** over the system.

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	

- Problems
 - 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?
 - such as issuing an I/O request to a disk, or gaining access to more system resources such as CPU or memory
 - how does the OS stop the process from running and switch to another process, thus implementing the time sharing?

Problem #1: Restricted Operations

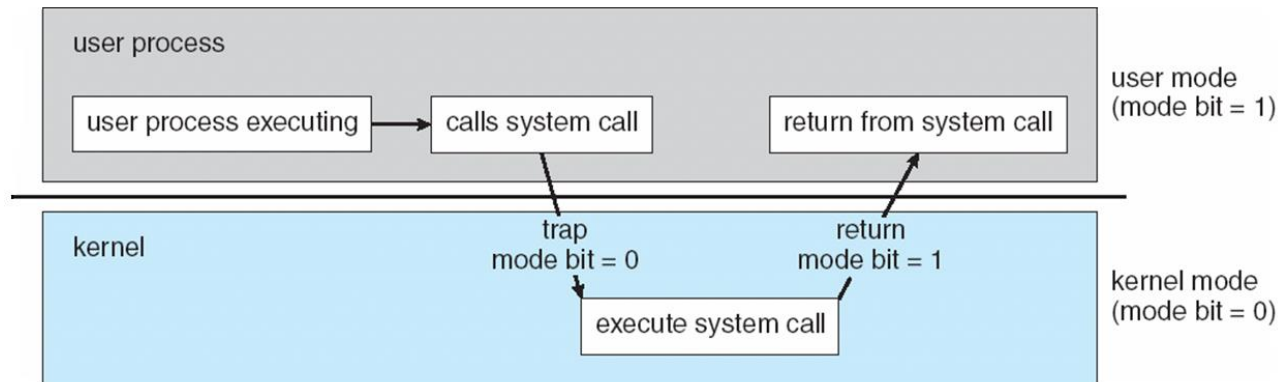
- Without limits on running programs, the OS wouldn't be in control of anything and thus would be "just a library"
- Two different privileged modes
 - user mode
 - code that runs in user mode is restricted in what it can do.
 - For example, when running in user mode, a process can't issue I/O requests; doing so would result in the processor raising an exception;
 - kernel mode
 - 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.

Problem #1: Restricted Operations

- System call
 - When a user process wishes to perform some kind of privileged operation
 - a program must execute a special **trap** instruction.
 - Trap must save the caller's registers in order to be able to return correctly to the caller
 - This instruction simultaneously jumps into the kernel and raises the privilege level to **kernel mode**.
 - User code specifies a **system-call number**
 - **Trap table** has the jump address for the system-call number
 - 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.

System Mode

- Dual-mode operation allows OS to protect itself and other system components
 - User mode and kernel mode (supervisor mode, privileged mode)
 - Mode bit provided by hardware
 - Some instructions designated as privileged, only executable in kernel mode
- Mode change
 - When a trap or interrupt occurs, the H/W switches from user mode to kernel mode



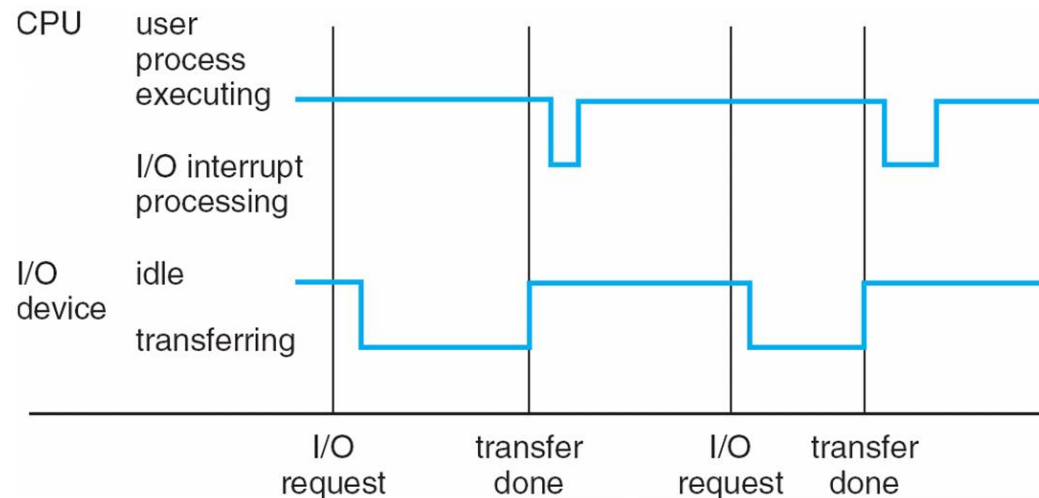
Interrupt and trap

- **Interrupt**

- Unexpected external event
 - I/O interrupt, Clock interrupt, Console interrupt
 - Machine check interrupt, Inter-process communication interrupt, etc
- From either the H/W or the S/W
 - H/W may trigger an interrupt by sending a signal to the CPU

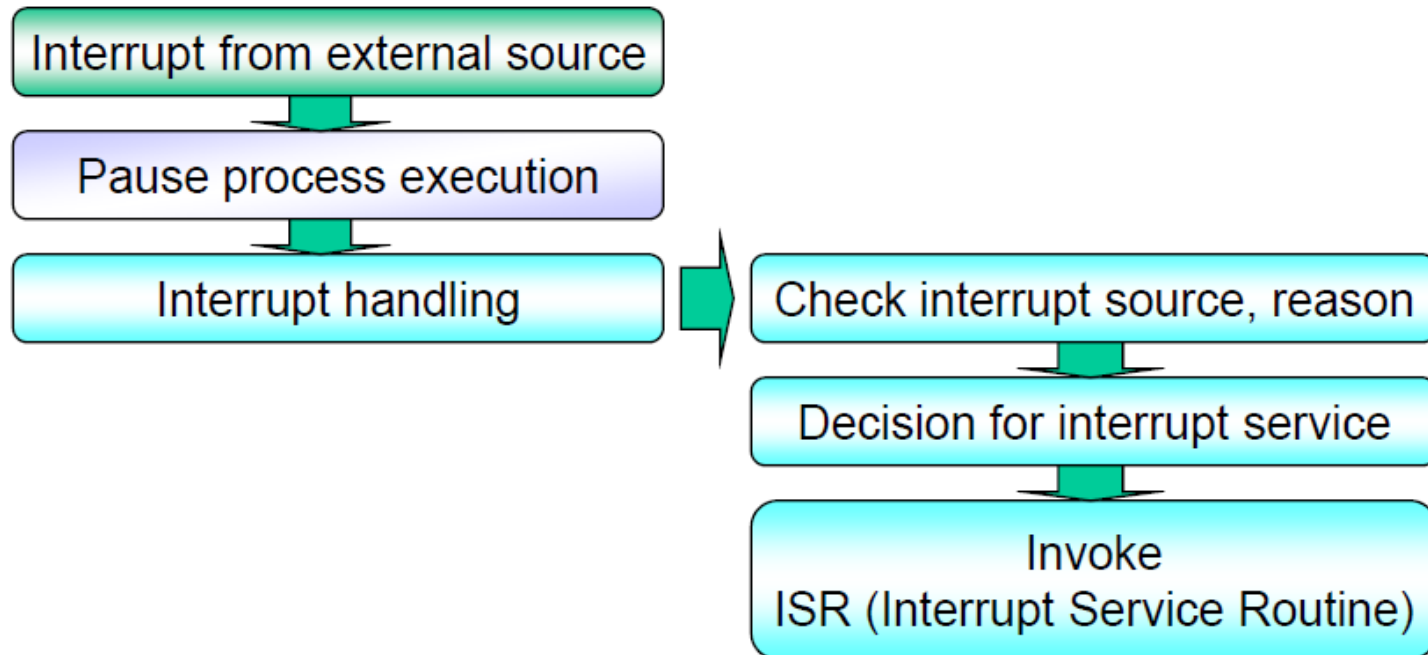
- **Trap**

- Software-generated interrupt caused by
 - **Exception**: An error of the running program (div by zero)
 - **System call**: A specific request from a user program:



Interrupts

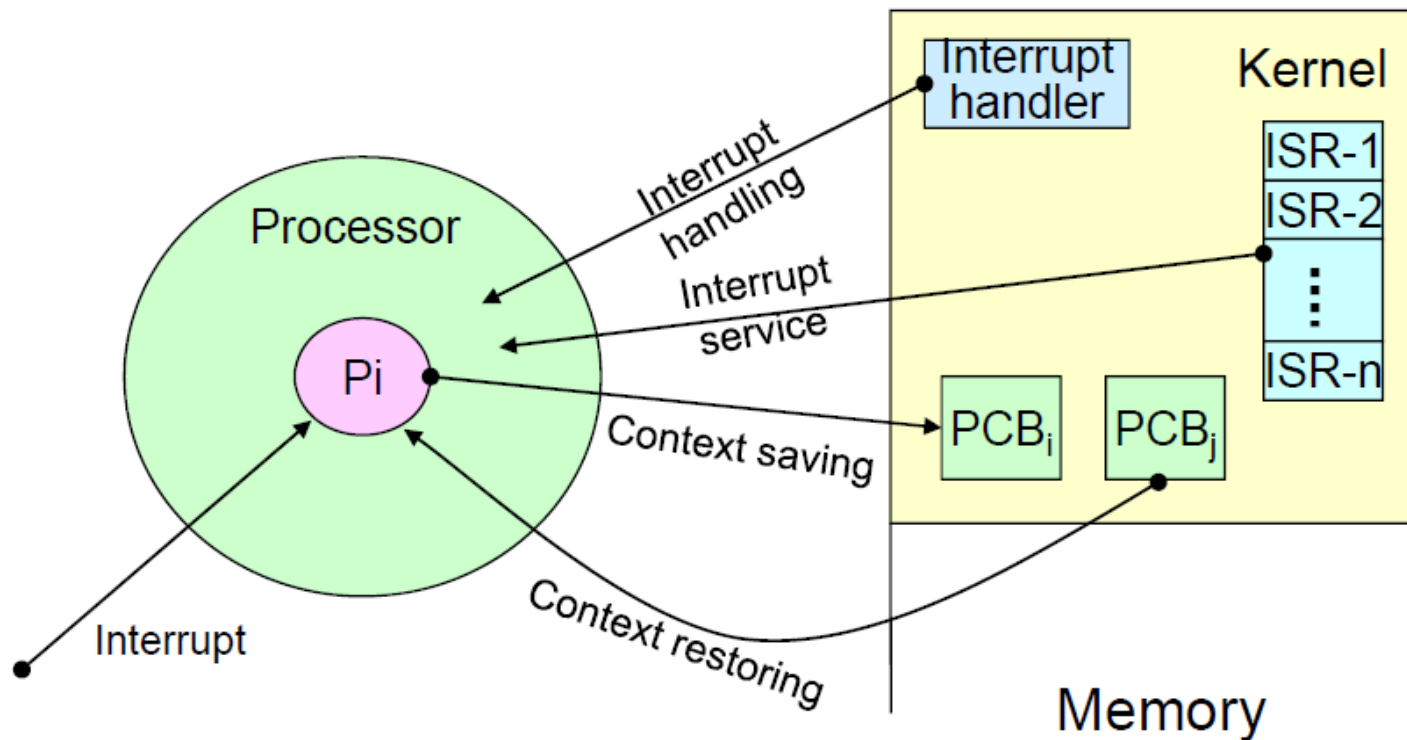
- **Interrupt handling process**



- ♦ kernel: Intervene whenever interrupt occurs and control interrupt handling and service process

Interrupts

Interrupt handling



Privileged instructions

- Machine instructions that can be executed only in kernel mode
- Examples
 - I/O control
 - Timer management
 - Interrupt management
 - Switching to user mode
 - Etc

Intel privileged instructions

CLTS - Clear Task-Switched Flag

HLT - Halt Processor

LGDT - Load GDT Register

LIDT - Load IDT Register

LLDT - Load LDT Register

LMSW - Load Machine Status

LTR - Load Task Register

MOV CRn - Move Control Register

MOV DRn - Move Debug Register

MOV TRn - Move Test Register

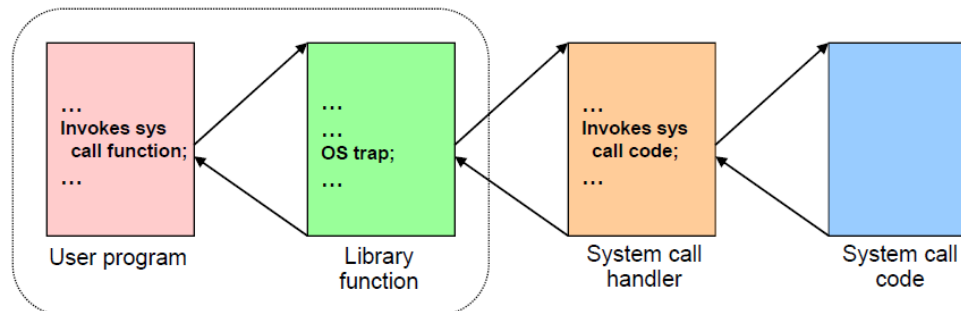
System Call Interface

- Interface between applications/processes and the OS
- Services that OS provides to applications/processes
 - Means for a user program to ask the OS to perform tasks reserved for OS on the user program's behalf
 - Means used by a process to request action by the OS
- Generally, programs use a high-level **Application Programming Interface** (API) rather than direct system call
 - POSIX API
- Executed by a generic **trap** instruction
 - **syscall** instruction in MIPS R2000
 - **swi** instruction in ARM
 - **INT/SYSENTER** in x86

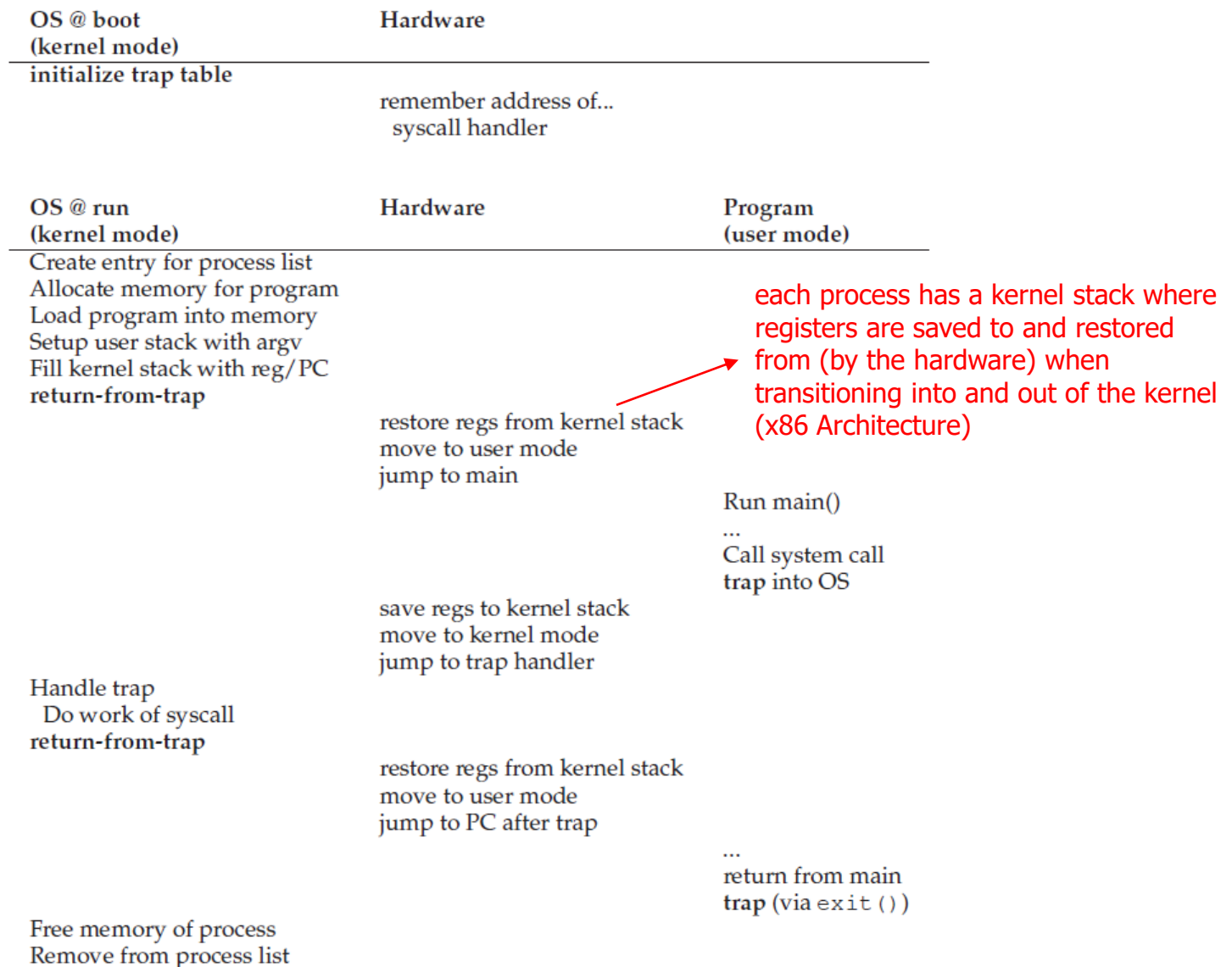
```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count)
```

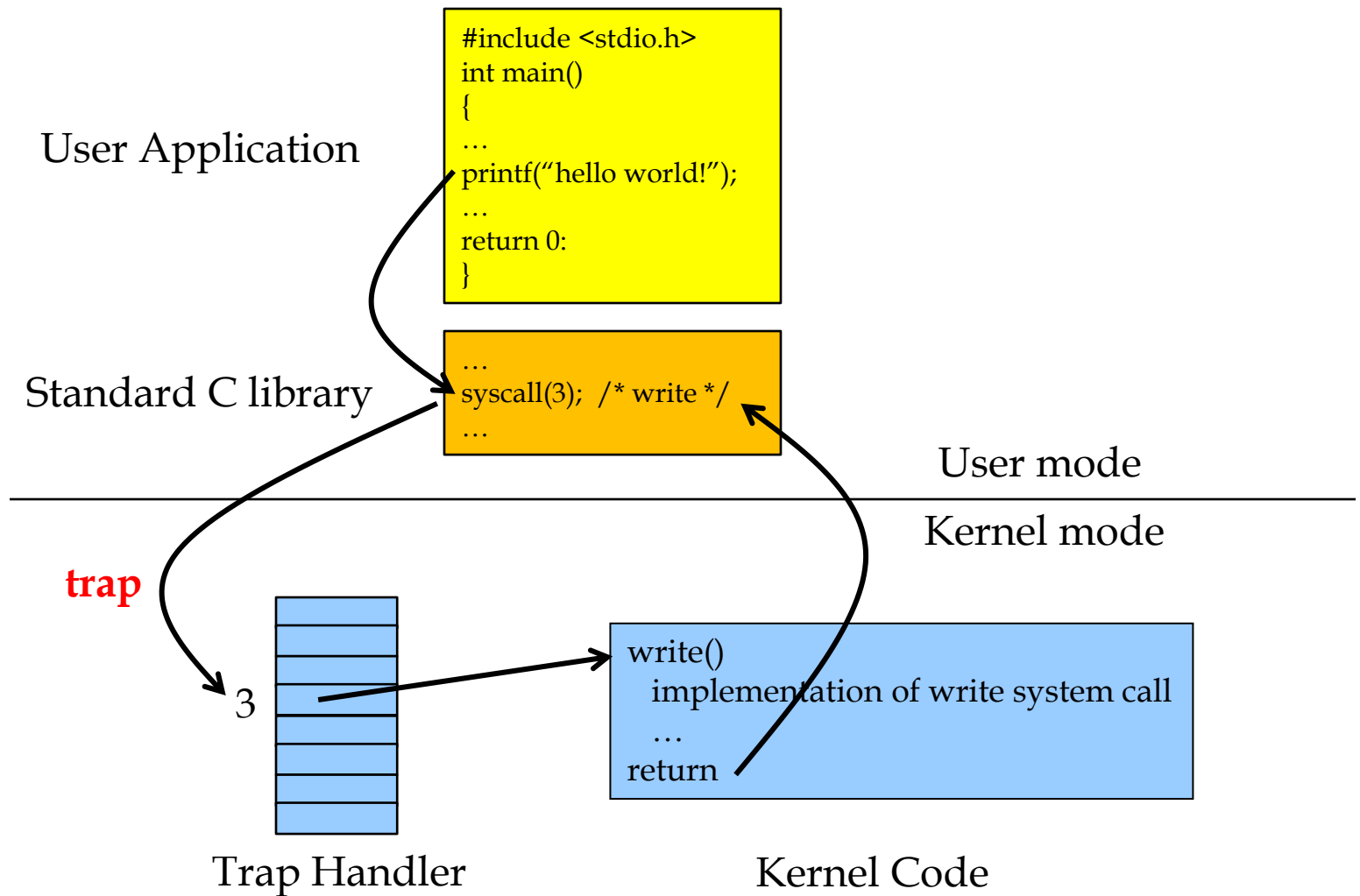
return value function name parameters



Limited Direct Execution



API – System Call – OS Relationship



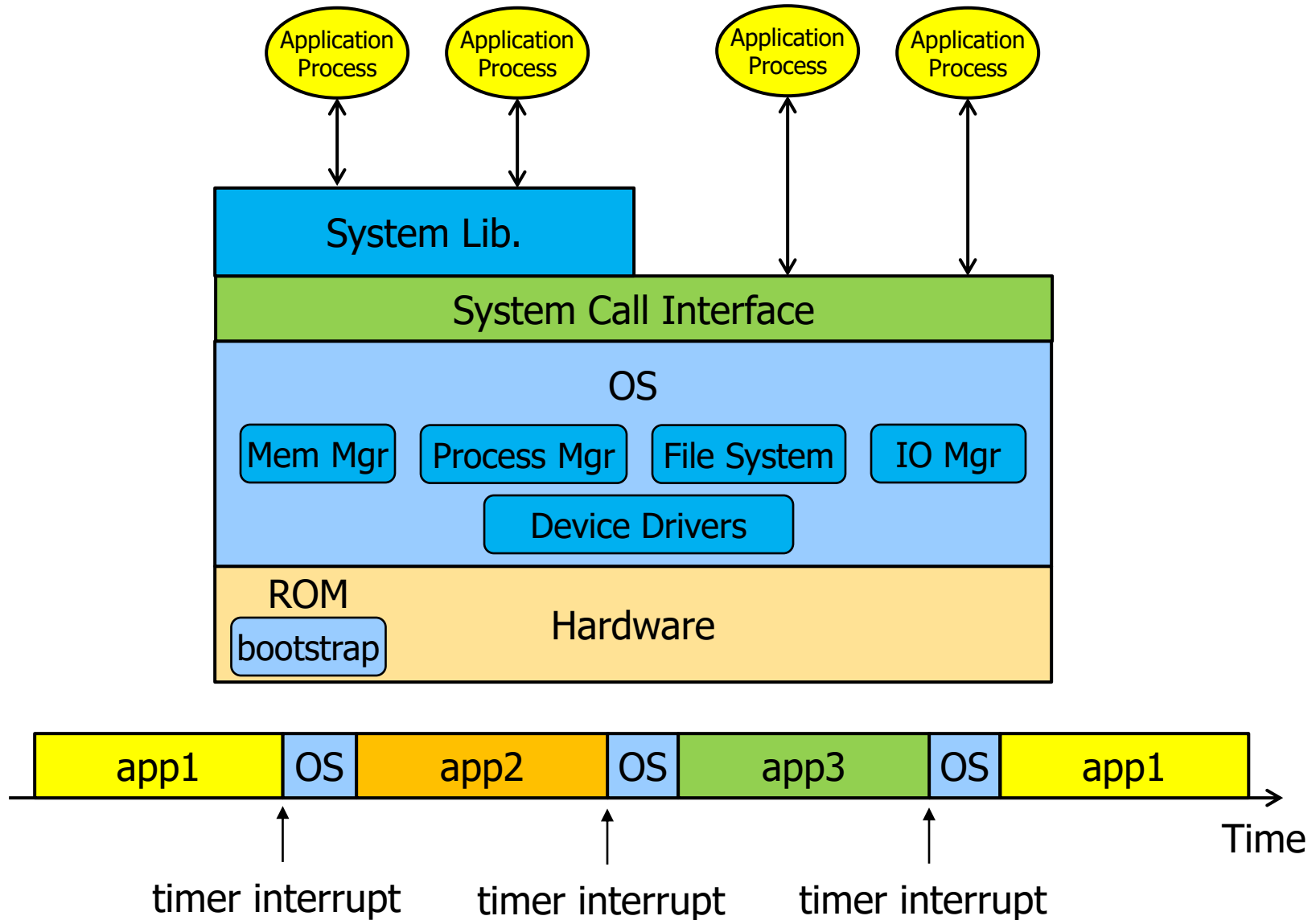
Problem #2: Switching Between Processes

- How can the OS **regain control** of the CPU so that it can switch between processes?
- **A Cooperative Approach: Wait For System Calls**
 - 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.
 - via system calls (including explicit **yield**), trap by exception
 - Passive approach
 - What happens if a process (whether malicious, or just full of bugs) ends up in an infinite loop, and never makes a system call?

Problem #2: Switching Between Processes

- **A Non-Cooperative Approach: The OS Takes Control**
 - **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.
 - save the state of the program that was running when the interrupt occurred
 - 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.

Problem #2: Switching Between Processes



Problem #2: Switching Between Processes

- When OS has regained control, it must decide whether to continue running the currently-running process, or switch to a different one
→ need **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**
- 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).
- 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.

Problem #2: Switching Between Processes

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
		...
	<p>timer interrupt</p> <p>save regs(A) to k-stack(A)</p> <p>move to kernel mode</p> <p>jump to trap handler</p>	
<p>Handle the trap</p> <p>Call switch() routine</p> <p>save regs(A) to proc-struct(A)</p> <p>restore regs(B) from proc-struct(B)</p> <p>switch to k-stack(B)</p> <p>return-from-trap (into B)</p>	<p>restore regs(B) from k-stack(B)</p> <p>move to user mode</p> <p>jump to B's PC</p>	
		Process B
		...

implicitly saved by H/W
(user register)

timer interrupt
save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler

How long ? {

Handle the trap

Call switch() routine

save regs(A) to proc-struct(A)

restore regs(B) from proc-struct(B)

switch to k-stack(B)

return-from-trap (into B)

explicitly saved by OS
(kernel register)

restore regs(B) from k-stack(B)

move to user mode

jump to B's PC

Context Switching

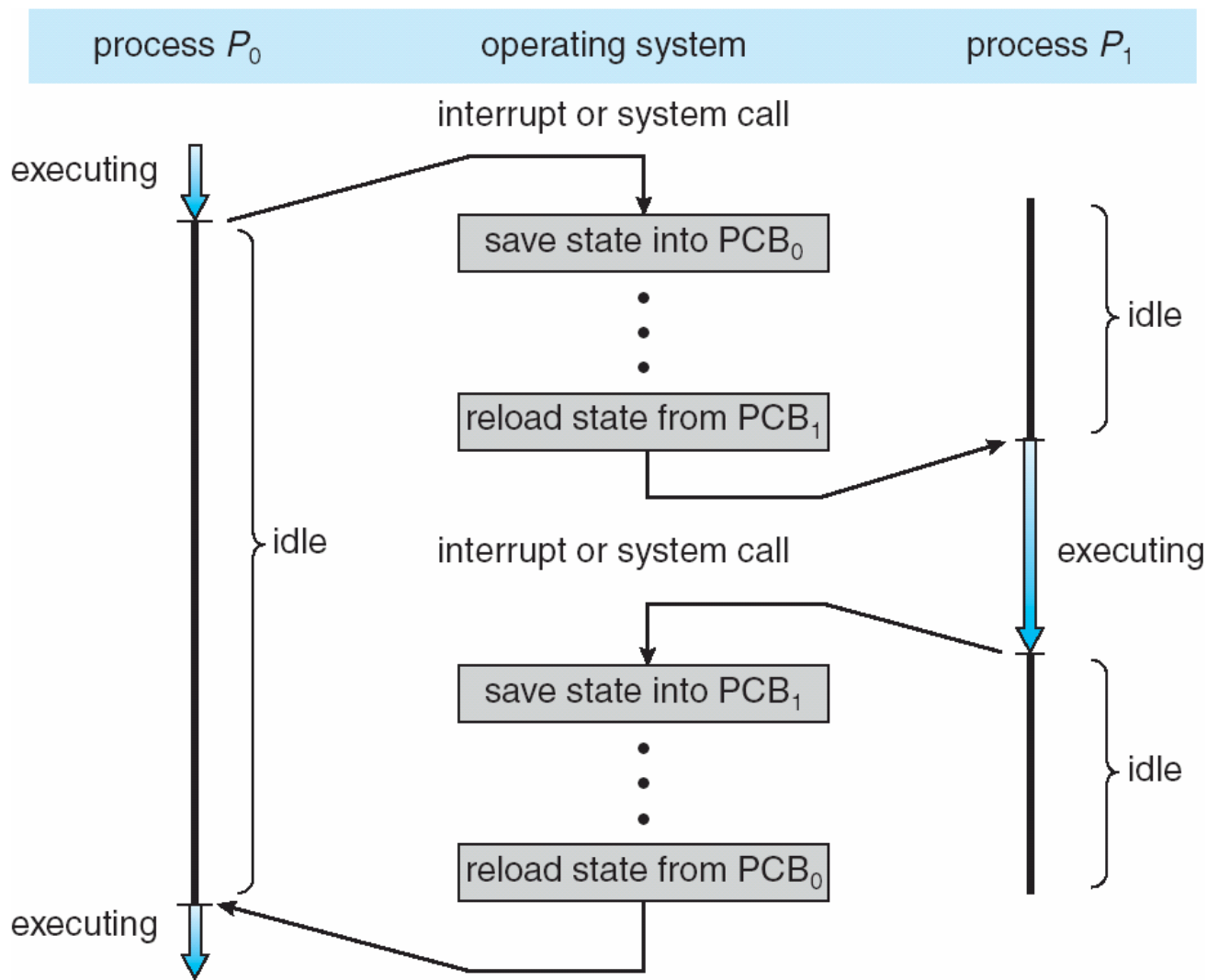
- **Context switch**

- Context saving: Saving the register context of a process when the process releases processor
- Context restoring: Reloading the register context of a process to continue execution
- Context switching: Saving the context of a process and restoring the context of another process

- **Context switch time**

- Pure overhead
 - The more complex the OS and the PCB → the longer the context switch
- Highly dependent on hardware support
 - Some processors provide multiple sets of registers
 - Context switch by changing the pointer to the current register set
 - Sun UltraSPARC, ARM

Context Switching



Worried About Concurrency?

- **Concurrency Issues**

- Interrupt during trap handling – “What happens when, during a system call, a timer interrupt occurs?”
- Interrupt during interrupt handling – “What happens when you’re handling one interrupt and another one happens?”
- One simple solution is **disable interrupts** during interrupt processing
 - disabling interrupts for too long could lead to lost interrupts
- More sophisticated approach is **locking**
 - protect concurrent access to internal data structures.
 - enables multiple activities to be on-going within the kernel at the same time, particularly useful on multiprocessors