Virtualization: The CPU

Mechanism: Limited direct execution

OSTEP Chapter 4+6:
http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-intro.pdf
http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf

Jan Reineke
Universität des Saarlandes

What is a process?

Process = running program

Process state =

Everything that may influence the execution of the process:

– PC (program counter) and other registers

– Address space: data + code

– Open files

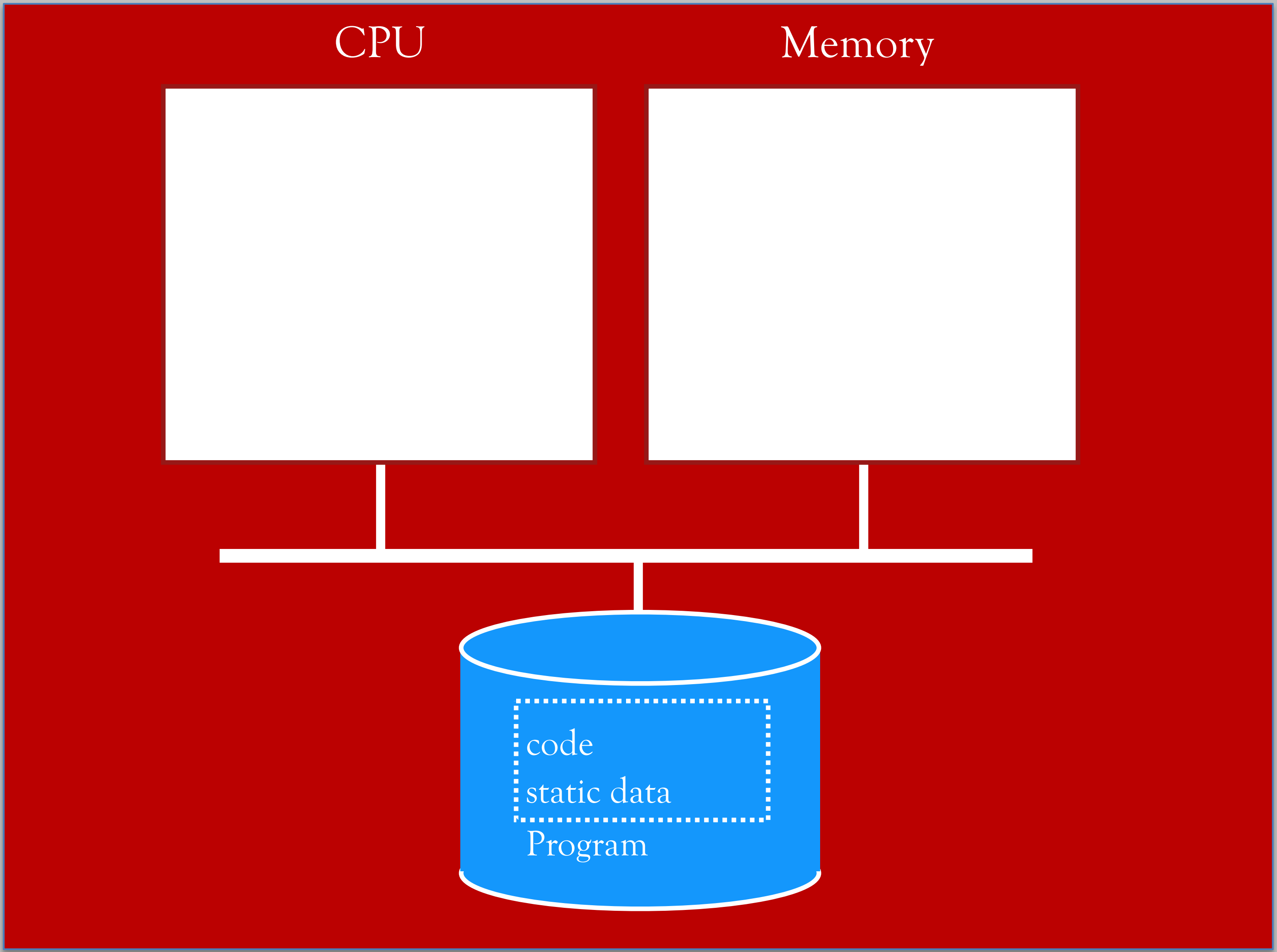
Processes versus Programs

Process ≠ Program:

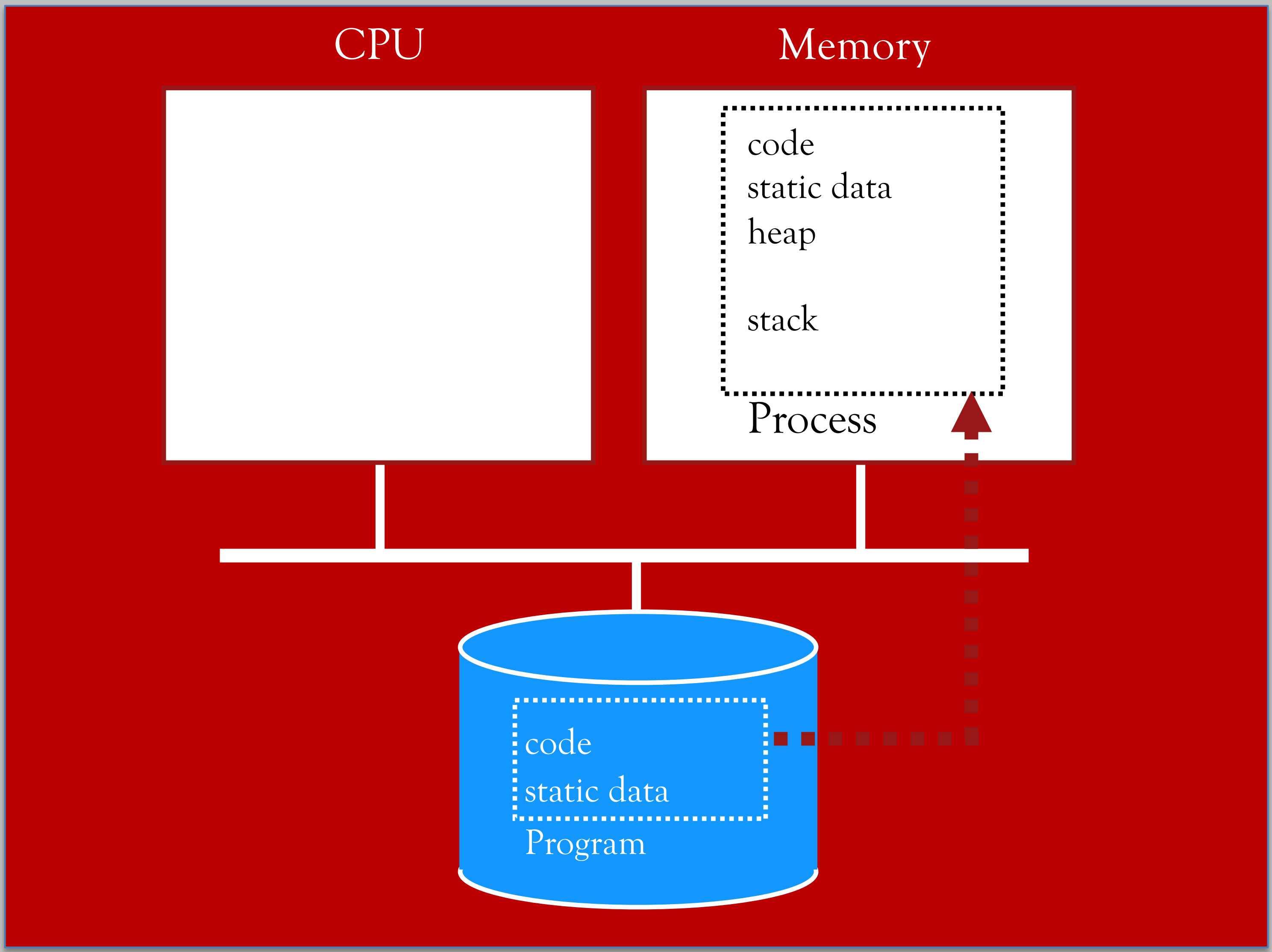
- *Program*: static code and static data
- *Process*: dynamic instance of code and data

Can have multiple process instances of same program.

Process creation



Process creation



Virtualizing the CPU

Goal:
Give each process impression it alone is actively using the CPU

Approach:
Partition resources in **time** and **space:**

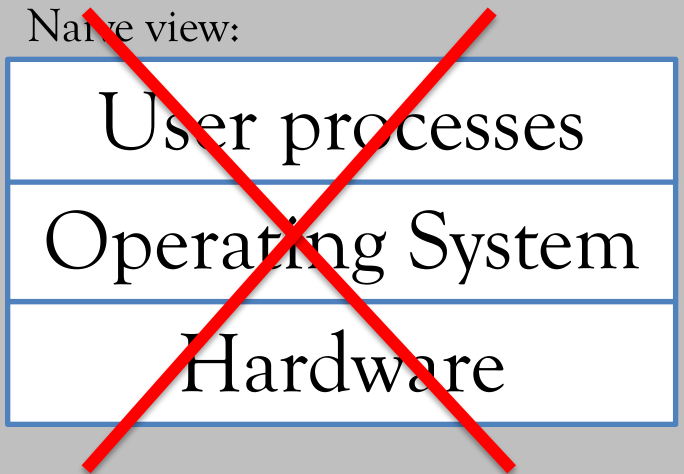
- *Processor*: partitioned in time
- *Memory*, *Disk*: partitioned in space (*later*)

How to provide good CPU performance?

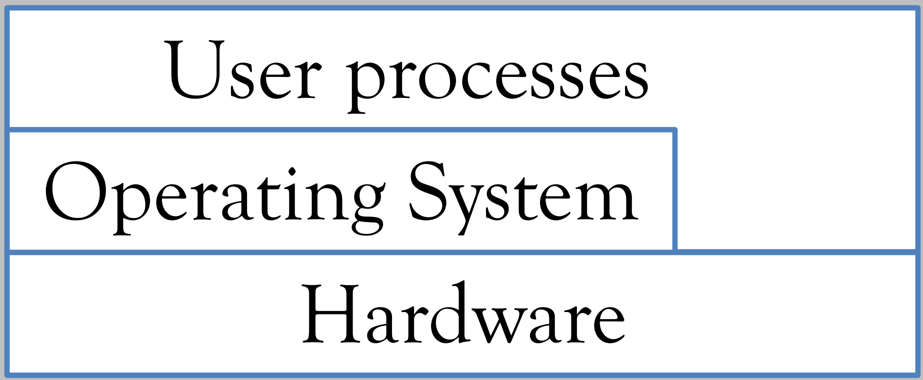
Direct execution:

- User processes are run **directly** on hardware
- OS creates process and transfers control to starting point (i.e., main())

Naive view:



More realistic:



How to provide good CPU performance?

Problems with **direct execution**?

1. Process wants to perform restricted operation
   → 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

Problem 1: Restricted operations

How can processes be executed in a **safe** *manner*
and *perform* **restricted operations***?*

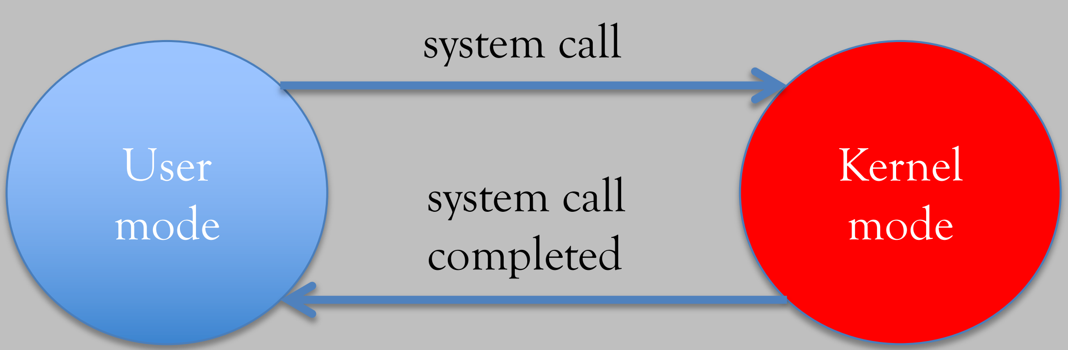
Solution: two (or more) execution modes

- User process run in **user mode** = restricted mode
  → no execution of restricted operations
- OS runs in **kernel mode** = unrestricted mode

Problem 1: Restricted operations

How can user process still perform restricted operations?

System calls = Function call into OS
+ change of privilege level



Example: System call

P would like to access I/O device via memory-mapped I/O (*more on this later*).

Example: System call

In **user mode** P can only see its own address space
(Address space of OS and other processes is hidden)

Example: System call

P can ask the operating system via a **system call** to access the I/O device:

MIPS code:
```
100: li $a0, 1234
104: li $v0, 1
108: syscall
```

MIPS convention:
Number of system call is written into $v0, parameter values in $a0.

Example: System call

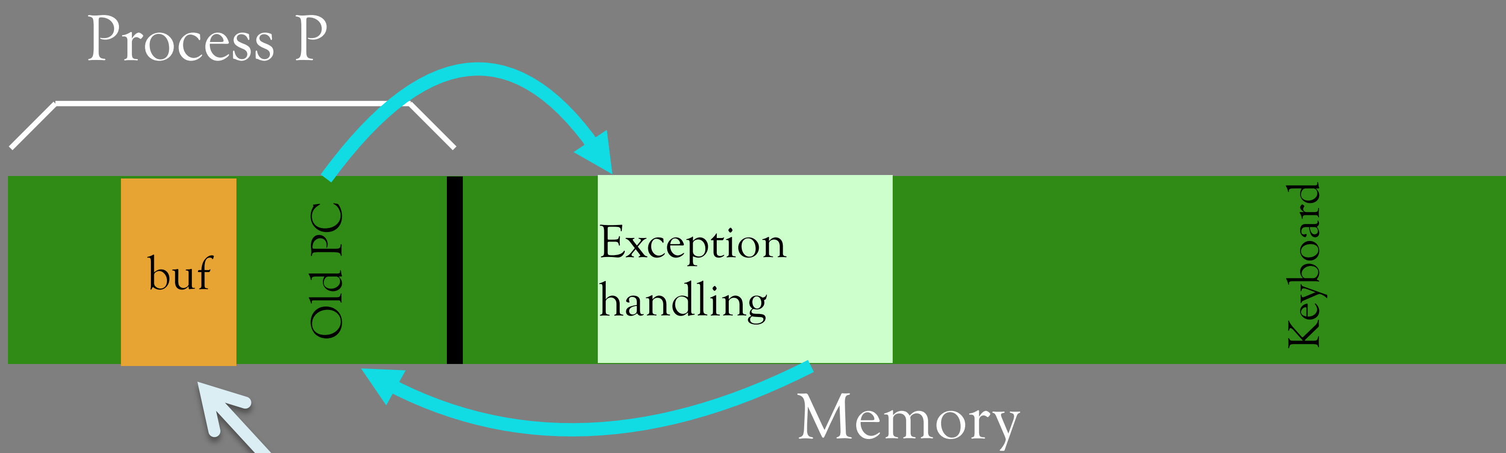
MIPS code:

```
100: li $a0, 1234
104: li $v0, 1
108: syscall
```

General flow upon system calls:

1. Switch to kernel mode
2. PC (and possibly other registers) saved by HW
3. PC is set to address in OS
4. System call is detected and processed
5. Switch back into user mode
6. Old PC+4 is reestablished and P resumes.

Example: System call



MIPS code:

```
100: li $a0, 1234
104: li $v0, 1
108: syscall
```

General flow upon system calls:

1. Switch to kernel mode
2. PC (and possibly other registers) saved by HW
3. PC is set to address in OS
4. System call is detected and processed
5. Switch back into user mode
6. Old PC+4 is reestablished and P resumes.

Which operations should be restricted?

User processes are **not** allowed to perform

- General memory access
- Directly interact with I/O devices (e.g. hard disks or SSDs)

System calls check whether requested access is permitted or not.

How is exception handling initialized?

- Processor starts OS in kernel mode
  - OS initializes exception handling
  - Only then the first user process is started

Problem 2: How to take CPU away?

OS requirements for multitasking:

- **Mechanism**
  to switch between processes
- **Policy**
  to decide which process to schedule when

Mechanisms versus policies

General principle:
Separation of mechanism and policy

- **Policy**:
  Makes decisions to optimize performance metrics
- **Mechanism**:
  Low-level code that implements the decision

Mechanism for context switch

Question 2A: *How does the OS gain control?*

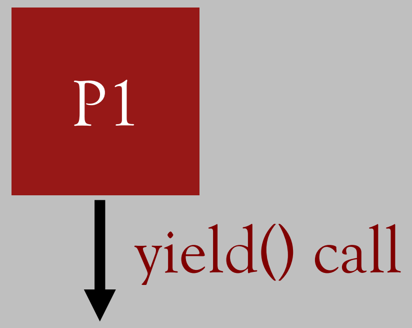
Question 2B: *What execution context must be saved restored upon context switch?*

Question 2A: How does the OS gain control?

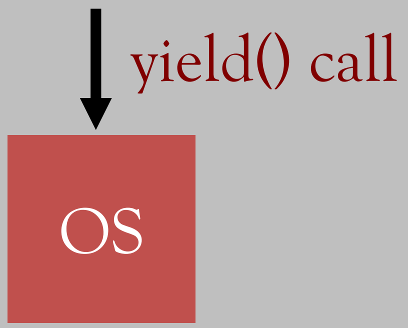
Option 1: Cooperative multitasking

- Trust user processes to relinquish CPU to OS
  - E.g. via system calls (or page faults (*later*))
  - Special system call `yield()`: offers OS to take over

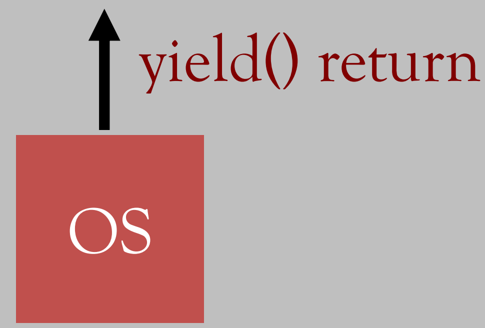
Cooperative approach



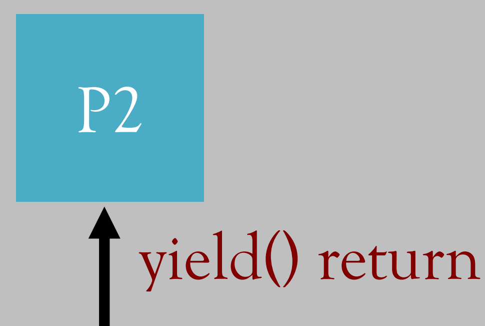
Cooperative approach



Cooperative approach



Cooperative approach



Cooperative approach

- **Disadvantage**: Process can "misbehave" and never relinquish control
  - only solution: reboot!

- Not performed in modern operating systems
  - Earlier in Windows 3.x, Mac OS 5 to 9, Atari ST

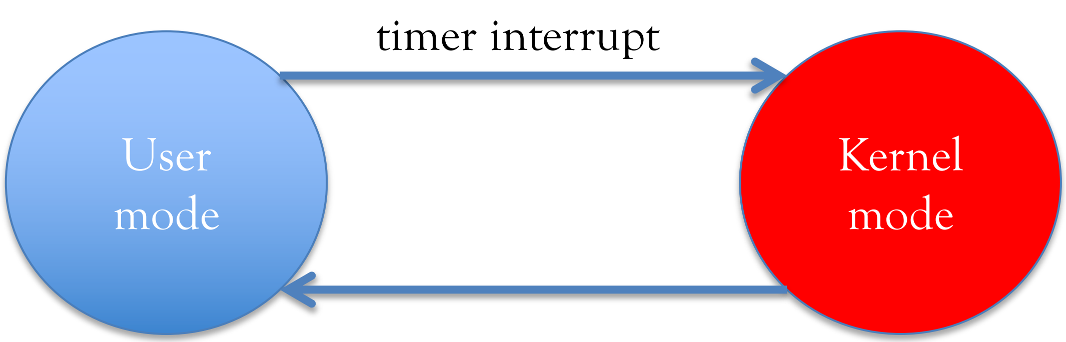
Question 2A: How does the OS gain control?

Option 2: Preemptive (true) multitasking

Periodic execution of OS code via hardware support:

– Hardware generates timer interrupt

– User processes cannot prevent these interrupt



Question 2B: What context must be saved?

Save context in process control block (PCB).

What information is stored in PCB?

- Process ID (PID)
- Process state (i.e., running, ready, or blocked)
- Registers, PC
- Pointers to open files
- Memory contents?

Operating system	Hardware	Program
		process A
		...

Flow on MIPS system

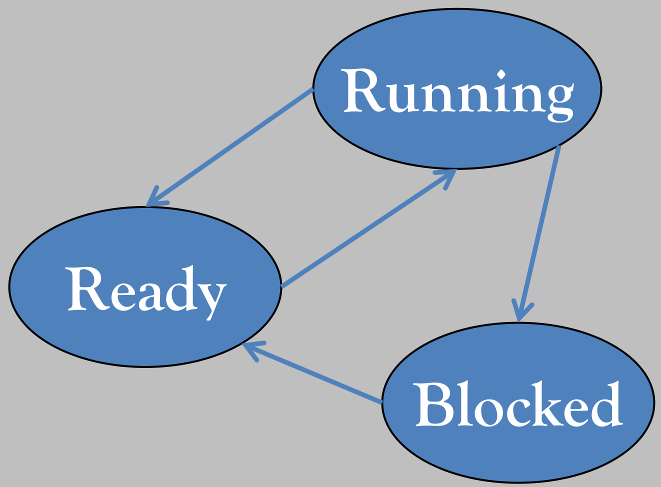
Operating system	Hardware	Program
		Process A ...
On x86 systems **hardware** (not the OS) saves the registers in memory	Timer interrupt: 1. Move to kernel mode 2. Save current PC in epc register 3. Set cause register 4. Write address of exception handling code into PC	
Process the exception: 1. Detect timer interrupt via cause register 2. Save registers of Process A in its process control block 3. Decide which process to execute next 4. Copy data from process control block of Process B into registers and set epc register 5. Call **eret** ("exception return")		
	1. Switch to user mode 2. Copy epc into PC	
		Process B ...

Problem 3: Slow operations such as I/O

When running process performs op that does not use CPU, OS switches to process that needs CPU.

OS tracks **mode** of each process:

- *Running*: on the CPU
- *Ready*: waiting for the CPU
- *Blocked*: Waiting for I/O or synchronization



Problem 3: Slow operations such as I/O

OS maintains several queues:

- *Ready queue:*
  contains all processes in mode “ready”
- *Event queue:* one queue per event:
  - e.g. disk I/O and locks
  - Contains all processes waiting for that event to complete

Policy determines which ready process to run

(*next lecture*)

Summary: Limited direct execution

- Direct execution makes processes fast
- *Problem 1:* Restricted operations
  Solution: User and kernel mode + System calls
- *Problem 2:* Multitasking
  Solution: Non-cooperative via timer interrupts
- *Problem 3:* Slow operations such as I/O
  Solution: OS runs those processes that are currently not waiting for I/O