

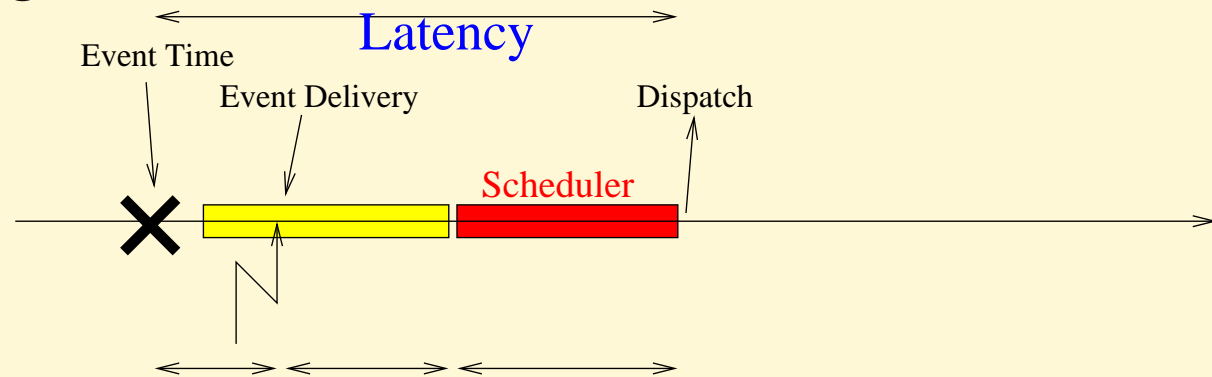
The OS Kernel

Luca Abeni

luca.abeni@santannapisa.it

Remember?

- Scheduler → triggered by internal (IPC, signal, ...) or external (IRQ) events
- Time between the triggering event and dispatch:
 - Event generation
 - Event delivery (interrupts may be disabled)
 - Scheduler activation (nonpreemptable sections)
 - Scheduling time



Kernel Latency!

Latency: Why?

- In real world, high priority tasks often suffer from blocking times coming from the OS (more precisely, from the kernel)
 - Why?
 - How?
 - What can we do?
- To answer the previous questions, we need to recall how the hardware and the OS work...

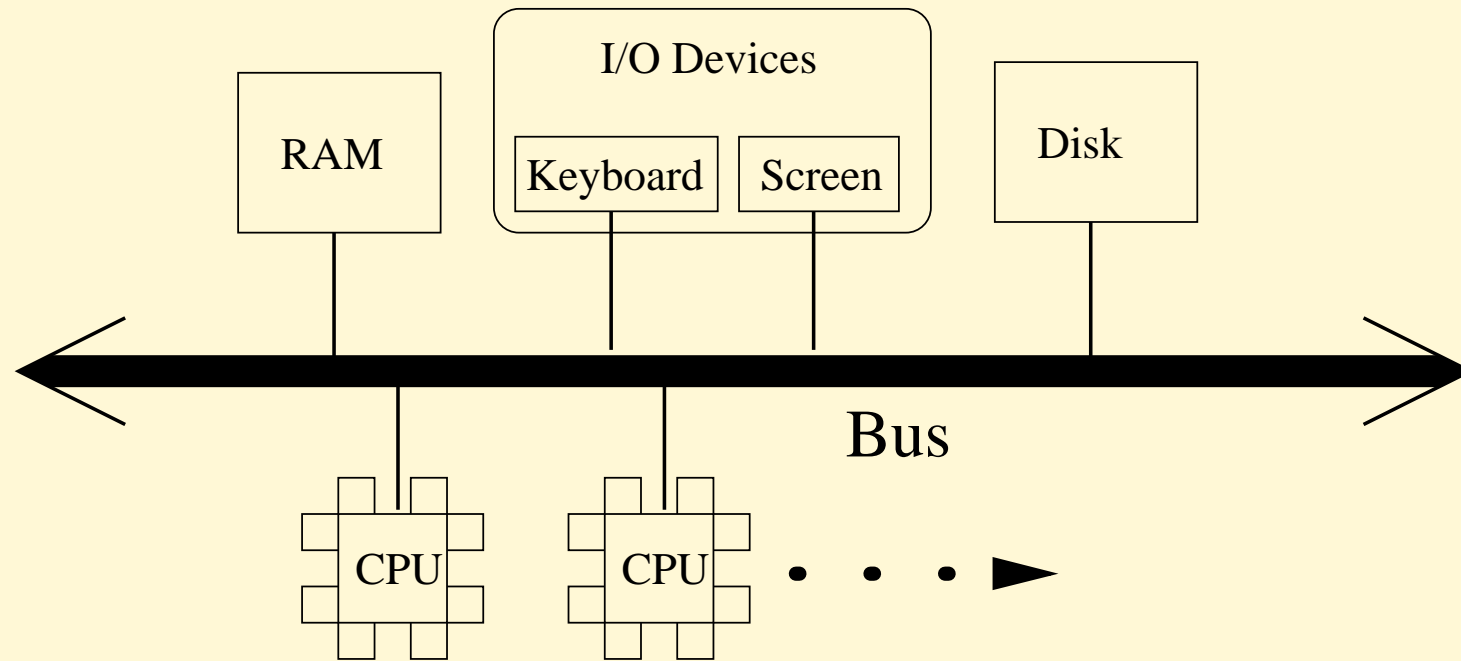
Computer Architecture - I

- A computer is composed of at least:
 - A **processor** (CPU)
 - Executes machine instructions
 - Might move data from / to memory
 - A **main memory** (RAM)
 - Used to store data and code (sequences of machine instructions)
 - Fast, but **volatile** (not persistent)
 - Some **storage memory**
 - **Slower** than RAM, but **persistent**
 - Some additional **input output devices** (I/O devices)

Computer Architecture - II

- All the components (one or more CPUs, RAM, I/O devices, ...) are connected by a **bus**
 - Example: system bus
 - Set of electrical connections
- Used to move data and code between CPU and RAM...
- ...or for Input and Output from / to devices or storage

Von Neumann Architecture



- Same memory containing both code and data
- Single bus connecting CPU, RAM and I / O devices

The CPU

- Fetches machine instructions from memory and executes them
 - Execution: might access memory (write / read data)
- Processing unit and control unit
 - Control unit: fetches the machine instructions
 - Processing unit (Arithmetic Logic Unit - ALU): executes the (arithmetic and logic) machine instructions
 - Modern CPUs: more units (FPU and others...)
- Contains some **registers**
 - Can be accessed by user code or not (invisible / hidden registers)

CPU Registers

- Invisible / hidden (cannot be referenced by machine instructions):
 - **Address Register** (AR): address we want to access on the bus
 - **Data Register** (DR): data to be written to / read from the bus
- Visible (referenced from machine instructions):
 - **Program Counter** (PC) / IP (Instruction Pointer): address of the next machine instruction to be executed
 - **Status Register** (SR) / F (Flags register): set of flags describing the machine state
 - Some data and address registers

Executing a Machine Instruction

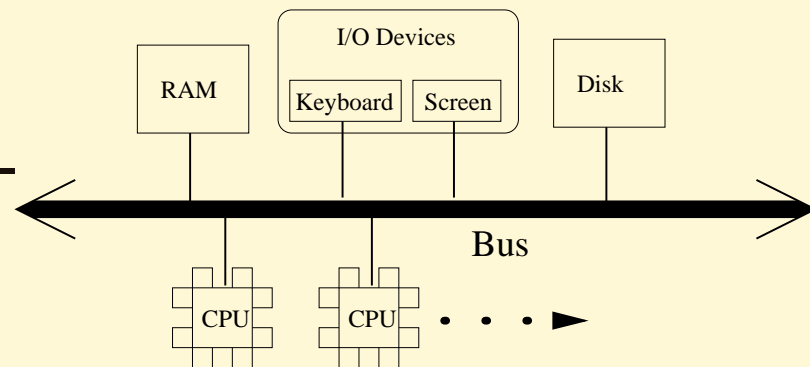
- Fetch the machine instruction to be executed
 - Copy PC into AR
 - Transfer data (indicated by AR) from RAM to DR
 - Save DR into an invisible register (instruction register)
 - Increase PC
- Decode: interpret the instruction saved in the instruction register
- Execute: perform the actions corresponding to the decoded instruction
 - If memory read, set AR, read DR, etc...
 - If memory write, set AR, write DR, etc...
 - Can modify PC (jump, etc...)

The Main Memory

- Von Neumann → The same memory contains both data and machine instructions
- Accessed through the bus
- Set of cells (locations) composed of 8 bit each
- Memory Access:
 - Load in AR the address of the cell to be accessed
 - If memory write, put the data in DR
 - Trigger the operation (read / write) on the bus
 - If memory read, get the data from DR

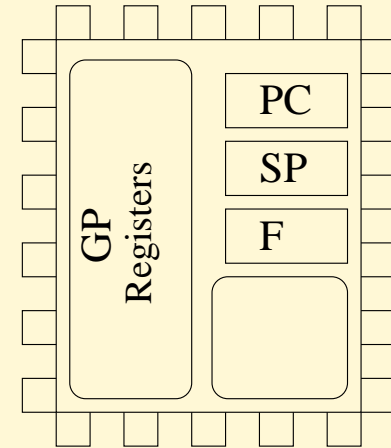
System Architecture

- System bus, interconnecting:
 - One or more CPU(s)
 - Memory (RAM)
 - I/O Devices
 - Secondary memory (disks, etc...)
 - Network cards
 - Graphic cards
 - Keyboard, mouse, etc



The CPU

- General-purpose registers
 - Can be accessed by all the programs
 - Sometimes, *data registers* or *address registers* instead of general-purpose



- Program Counter (PC) - AKA Instruction Pointer
- Stack Pointer (SP) register
- Flags register (AKA Program Status Word)
- Some “special” registers
 - Control how the CPU works, must be “protected”

The CPU - Protection

- Regular user programs should not be allowed to:
 - Influence the CPU mode of operation
 - Perform I/O operations
 - Reconfigure virtual memory
- ⇒ Need for “privileged” mode of execution
 - Regular registers vs “special” registers
 - Regular instructions vs privileged instructions
- User programs: low privilege level (*User Level*)
- The OS *kernel* runs in *Supervisor Mode*

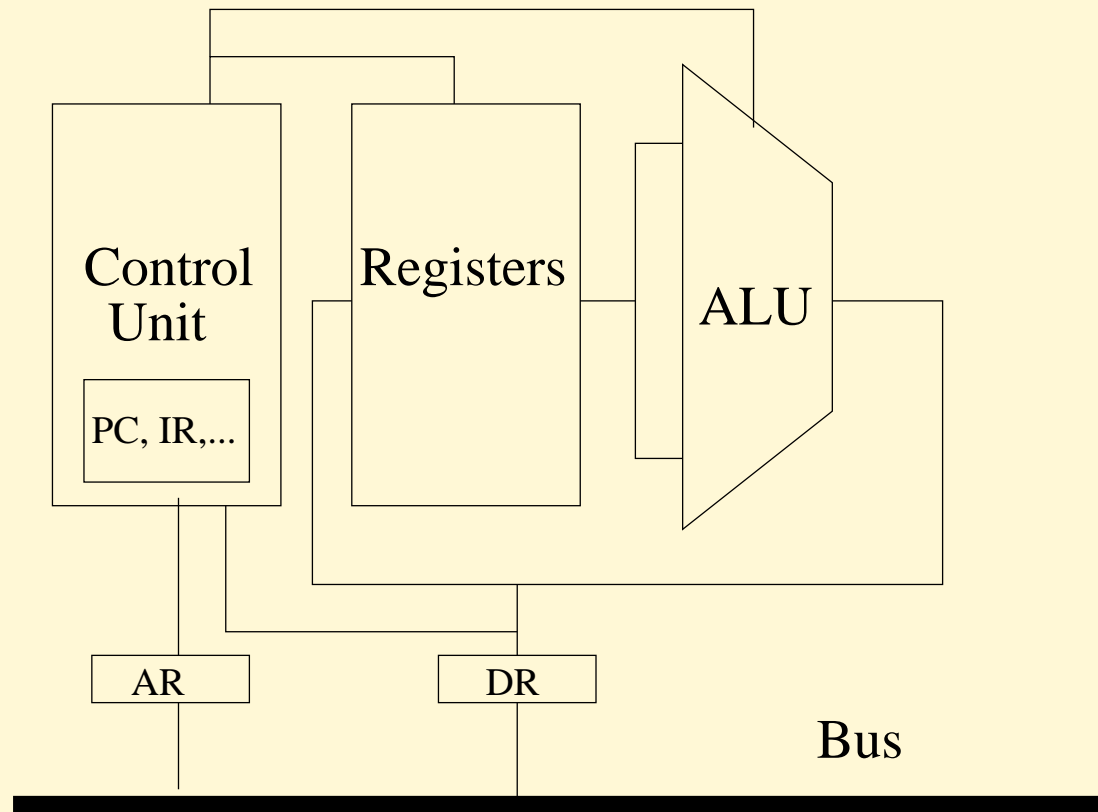
An Example: Intel x86

- Real CPUs are more complex. Example: Intel x86
 - Few GP registers: EAX, EBX, ECX, EDX (accumulator registers - containing an 8bit part and a 16bit part), EBP, ESI, EDI
 - EAX: Main accumulator
 - EBX: Sometimes used as base for arrays
 - ECX: Sometimes used as counter
 - EBP: Stack base pointer (for subroutines calls)
 - ESI: Source Index
 - EDI: Destination Index

Intel x86 - 2

- Segmented memory architecture
 - Segment registers CS (code segment), DS (data segment), SS (stack segment), GS, FS
- Various modes of operation: RM, PM, VM86, x86-64, ...
 - Mainly due to backward compatibility

Example of (Toy) CPU

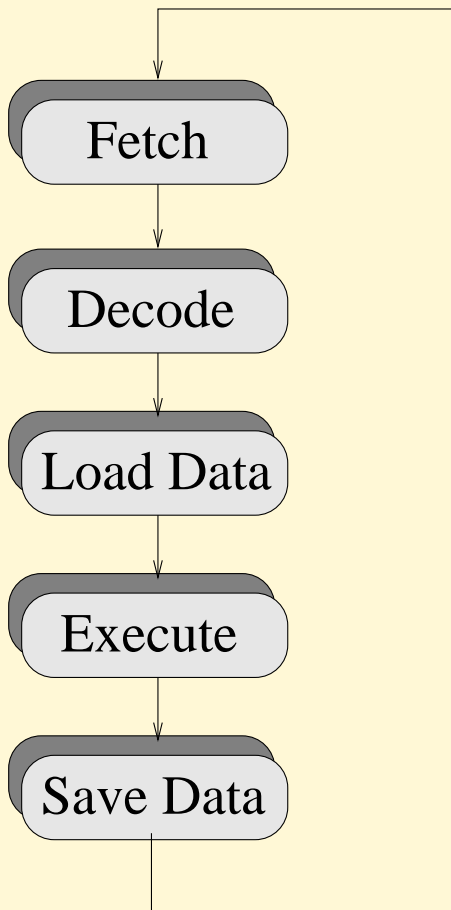


- Toy CPU: just an example with many simplifications
- Modern (real) CPUs are much more complex!
 - Pipeline
 - Parallel execution

CPUs, Programs, & Friends

- CPU → executes programs
 - Stored in main memory
 - Use data from main memory
- Program: formal description of an algorithm
 - Using a programming language
- Sequence of machine instructions
 - **Actions** having **effects** on some **objects**
 - “Object”: data stored in main memory
- Instance of program in execution: sequence of actions on objects
 - Example: `int mcd(int a, int b)` and its execution

Executing a Program



- CPU: cyclical execution (fetch / decode / load / execute / save)
 - Machine instructions are executed (mainly) sequentially
- Machine designed to execute its own language!
 - Machine Language

Physical Machines...

- Computer: (physical) machine designed to execute programs
- Every machine executes programs written in **its own language**
- Relationship between **machine** and **language**
 - A machine has its own language (the language it can parse and execute)
 - A language can be “understood” (parsed and executed) by multiple different machines
- Program execution: (infinite) cycle
fetch/decode/load/execute/save
 - CPU: hw implementation of this cycle

...And Abstract Machines!

- The fetch/decode/load/execute/save cycle can be implemented in hw or in sw...
- Software Implementation: **Abstract Machine**
 - Algorithms and data structures used to **store** and **execute** programs
- Once upon a time referred as “*Virtual Machine*”
 - Today, the term “Virtual Machine” (VM) is used with a slightly different meaning

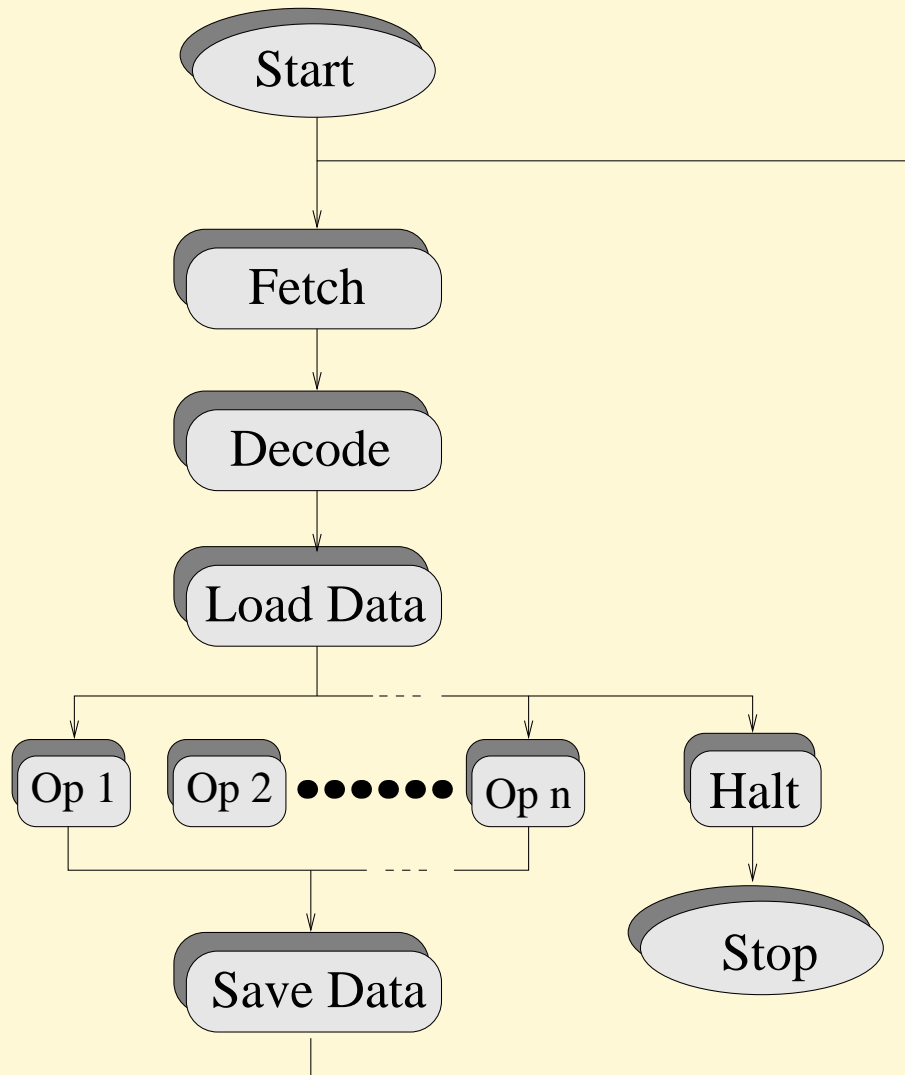
Abstract Machines and Languages

- Similarly to physical machines (CPUs), each abstract machine has its own machine language
 - Machine language for a CPU: sequence of 0 / 1
 - Assembly makes it more readable
 - Abstract machines generally have higher level machine languages (C, Java, etc...)
- $\mathcal{M}_{\mathcal{L}}$: abstract machine understanding language \mathcal{L}
 - \mathcal{L} is the *machine language* of $\mathcal{M}_{\mathcal{L}}$
 - Program: sequence of instructions written in \mathcal{L}
- $\mathcal{M}_{\mathcal{L}}$ is just a possible way to describe \mathcal{L}

Abstract Machines Behaviour

- To execute a program written in \mathcal{L} , $\mathcal{M}_{\mathcal{L}}$ has to:
 1. Execute some “elementary operations”
 - In hw, ALU
 2. Manage the execution flow
 - Execution is not only sequential (jumps, loops, etc...)
 - In hw, PC handling
 3. Move data from / to memory
 - Addressing modes, ...
 4. Take care of memory management
 - Dynamic allocation, stack management, etc...

Abstract Machine Example



- Execution cycle: very similar to a CPU...
- ... But it is implemented in software!

Multiple Flows of Instructions

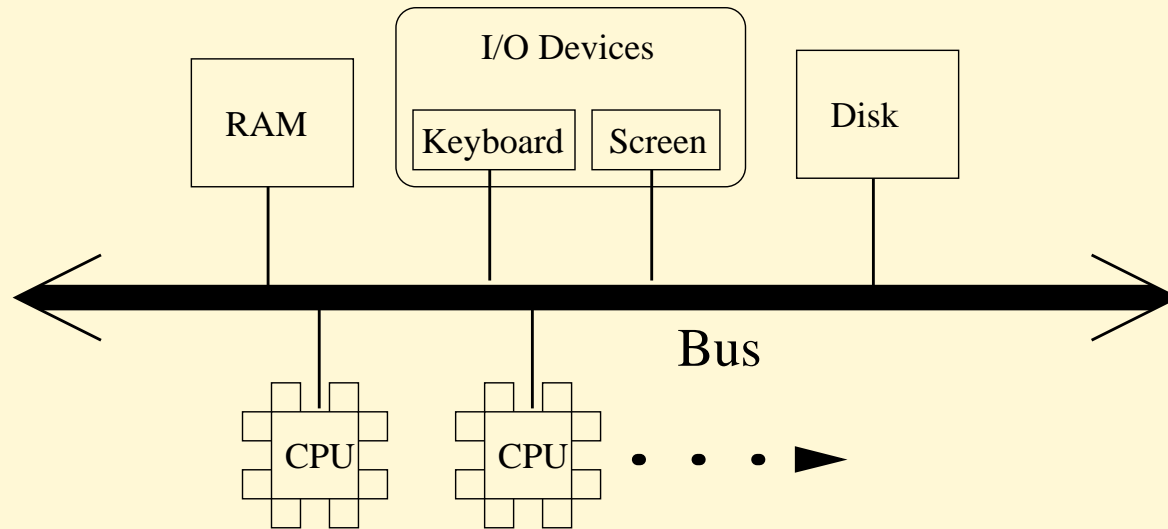
- A modern computer has at least a CPU...
- ...And each CPU is the hw implementation of an abstract machine
 - Abstract machine describing the whole computer?
 - Programs are not sequential anymore!!!
- An execution flow (fetch/decode/load/execute/save cycle) per CPU
- “Concurrent” machine model

Concurrent Machines

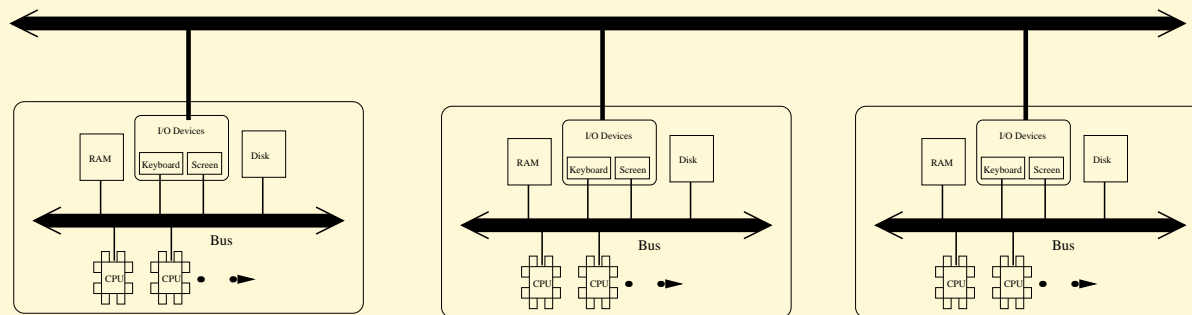
- Execute M instruction flows in parallel
 - Hardware implementation: $M = \text{number of CPUs} / \text{CPU cores}$
- Various possible architectures
 - Shared memory model (hw: SMP machines)
 - Private memory model (hw: network of M computing nodes)
 - Various trade-offs between the two (NUMA, etc...)
- Issue: the various flows are not independent
 - Concurrent accesses to memory?
 - Synchronization?

Concurrent Machine Architectures

- Shared memory



- Private memory



Concurrent Abstract Machines

- I said: “Abstract Machine \equiv *Algorithms and data structures used to store and execute programs*”
 - Is this correct when considering concurrent execution?
 - Yes! The “issue” is in the description of how to execute a program
- Single fetch/decode/load/execute/save cycle:
sequential program \Rightarrow Sequential Abstract Machine
- **Concurrent Abstract Machine**: can store and execute **concurrent programs**
 - Multiple, concurrent, execution cycles!
 - Machine language: concurrent language!

Concurrent Abstract Machine Architectures

- As for physical machines, various possible architectures
 - Shared memory (threads)
 - Private memory (processes)
 - Trade-offs (multi-threaded processes, processes sharing memory, ...)
- Result in different programming models
 - Shared resources with mutexes / condvars
 - Message passing
 - ...
- Different programming styles (cooperative resource management vs servers...)
- And different problems to be addressed

The OS as an Abstract Machine

- Concurrent Abstract Machine
 - Support for the execution of concurrent programs
 - Multiple execution flows
 - No relationship with the number of physical CPUs (or CPU cores)
 - Can have more execution flows than physical CPUs / CPU cores
- The Operating System implements this abstract machine
 - Machine language: the CPU machine language **augmented with system calls**

The Operating System

- Operating System: set of programs and libraries implementing the (concurrent) abstract machine
- In particular, the OS kernel implements:
 - Concurrency
 - Allows to execute multiple instruction flows on a smaller number of physical CPUs
 - Synchronization / Communication
 - Allows the multiple instruction flows to interact
 - Protection
 - Give exclusive access to some shared resources (example: memory) to some instruction flows

The Kernel

- Part of the OS which manages the hardware
- Runs with the CPU in *Supervisor Mode* (high privilege level)
 - Privilege level known as *Kernel Level* (KL) - execution in *Kernel Space*
 - Regular programs run in *User Space*
- Mechanisms for increasing the privilege level (from US to KS) **in a controlled way**
 - Interrupts (+ traps / hw exceptions)
 - Instructions causing a hardware exception

Interrupts and Hardware Exceptions

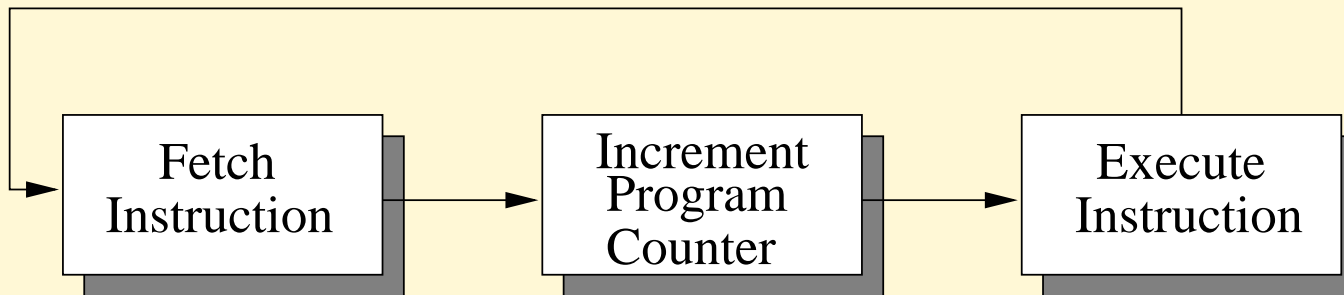
- Switch the CPU from User Level to Supervisor Mode
 - Enter the kernel
 - Can be used to implement *system calls*
- A partial Context Switch is performed
 - Flags and PC are pushed on the stack
 - If processor is executing at User Level, switch to Kernel Level, and eventually switch to a *kernel stack*
 - Execution jumps to a handler in the kernel → save the user registers for restoring them later

Back to User Space

- Return to low privilege level (execution returns to User Space) through a “return from interrupt” Assembly instruction (`IRET` on x86)
 - Pop flags and PC from the stack
 - Eventually switch back to user stack
- Return path for system calls and hardware interrupt handlers

Simplified CPU Execution

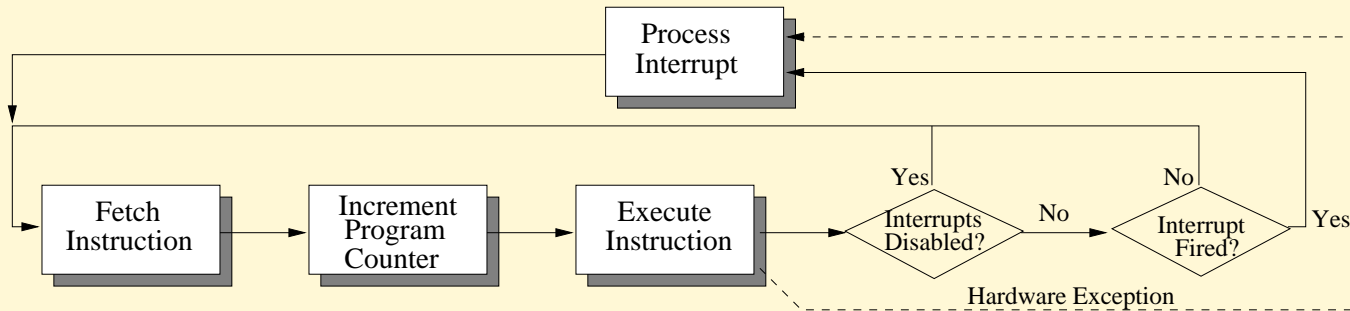
- To understand interrupts, consider simplified CPU execution first
 - Simplification respect to the fetch/decode/load/execute/save cycle



- The CPU iteratively:
 - Fetches an instruction (address given by PC)
 - Increases the PC
 - Executes the instruction (might update the PC on jump...)

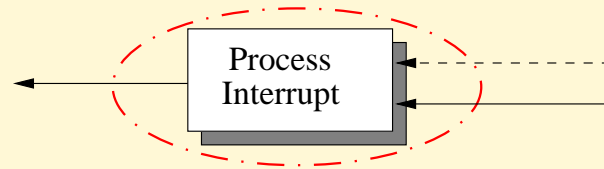
CPU Execution with Interrupts

- More realistic execution model



- Interrupt: cannot fire during the execution of an instruction
- Hardware exception: caused by the execution of an instruction
 - `trap`, `syscall`, `sc`, ...
 - I/O instructions at low privilege level, Page faults, ...

Processing Interrupts

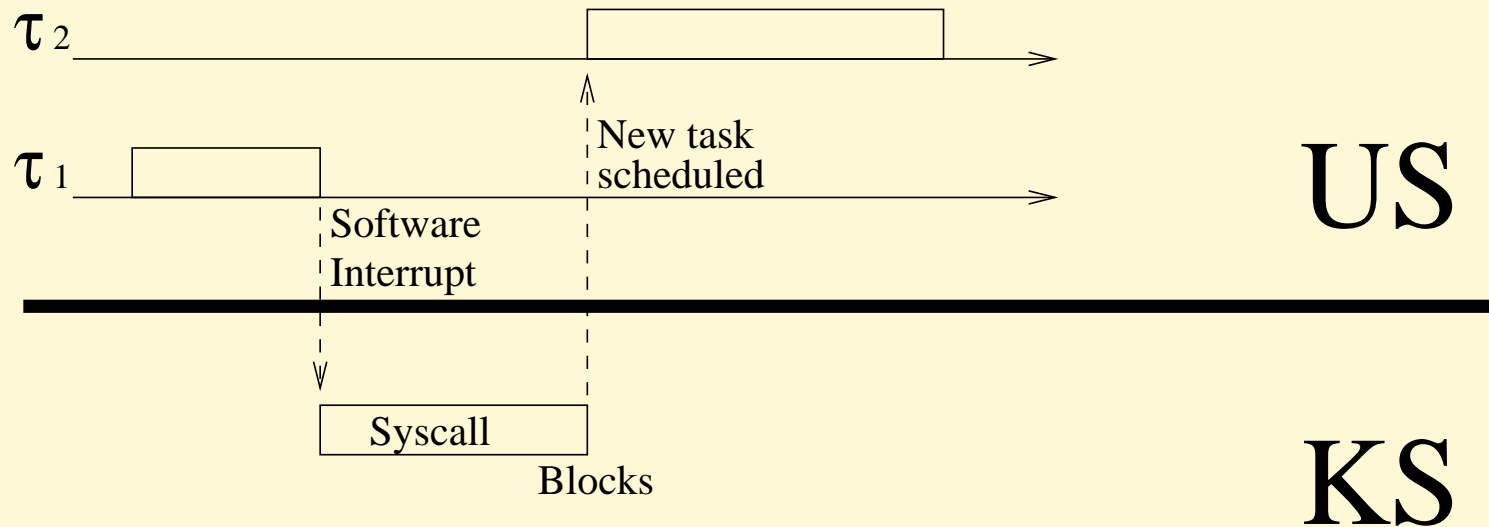


- *Interrupt table* → addresses of the handlers
 - Interrupt n fires \Rightarrow after eventually switching to KS and pushing flags and PC on the stack
 - Read the address contained in the n^{th} entry of the interrupt table, and jump to it!

Interrupt Tables

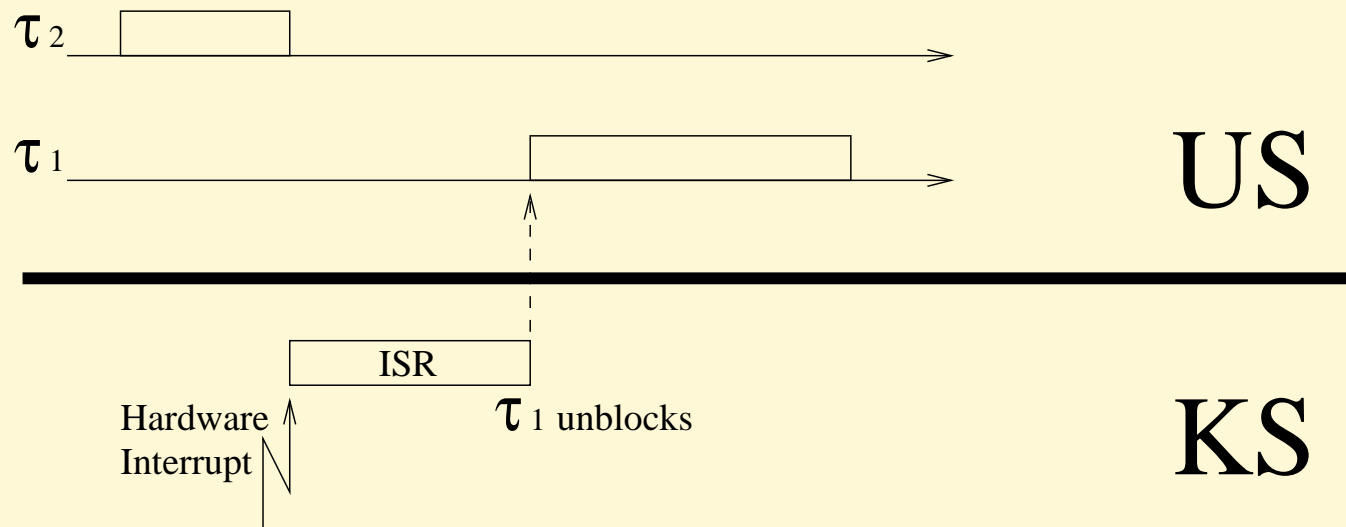
- Implemented in hardware or in software
 - x86 → **I**nterrupt **D**escription **T**able composed of interrupt gates. The CPU automatically jumps to the n^{th} interrupt gate
 - Other CPUs jump to a fixed address → a software demultiplexer reads the interrupt table

Software Interrupt - System Call



1. Task τ_1 executes and invokes a system call
2. Execution passes from US to KS (change stack, push PC & flags, increase privilege level)
3. The invoked syscall executes. Maybe, it is blocking
4. τ_1 blocks \rightarrow back to US, and τ_2 is scheduled

Hardware Interrupt



1. While τ_2 is executing, a hardware interrupt fires
2. Execution passes from US to KS (change stack, push PC & flags, increase privilege level)
3. The proper **I**nterrupt **S**ervice **R**outine executes
4. The ISR can unblock τ_1 \rightarrow when execution returns to US, τ_1 is scheduled

Summing up...

- The execution flow enters the kernel for two reasons:
 - Reacting to events “coming from up” (syscalls)
 - Reacting to an event “coming from below” (an hardware interrupt from a device)
- The kernel executes in the context of the interrupted task

Blocking / Waking up Tasks...

- A system call can block the invoking task, or can unblock a different task
- An ISR can unblock a task
- If a task is blocked / unblocked, when returning to user space a context switch can happen

The scheduler is invoked
when returning from KS to US

Example: I/O Operation

- Consider a generic Input or Output to an external device (example: a PCI card)
 - Performed by the kernel
 - User programs must use a syscall
- The operation is performed in 3 phases
 1. **Setup**: prepare the device for the I/O operation
 2. **Wait**: wait for the end of the operation
 3. **Cleanup**: complete the operation
- Can be done using polling, PIO, DMA, ...

Polling

- User programs invoke the kernel; execution in kernel space until the operation is terminated
- The kernel cyclically reads (polls) an interface status register to check if the operation is terminated
- Busy-waiting in kernel space!
 - No user task can execute while waiting for the I/O operation...
 - The operation **must** be very short!
 - I/O operation == blocking time

Polling - 2

1. The user program raises a software input
2. Setup phase - in kernel: in case of input operation, nothing is done; in case of output operation, write a value to a card register
3. Wait - in kernel: cycle until a bit of the card status register becomes 1
4. Cleanup - in kernel: in case of input, read a value from a card register; in case of output, nothing is done. Eventually return to phase 1
5. IRET

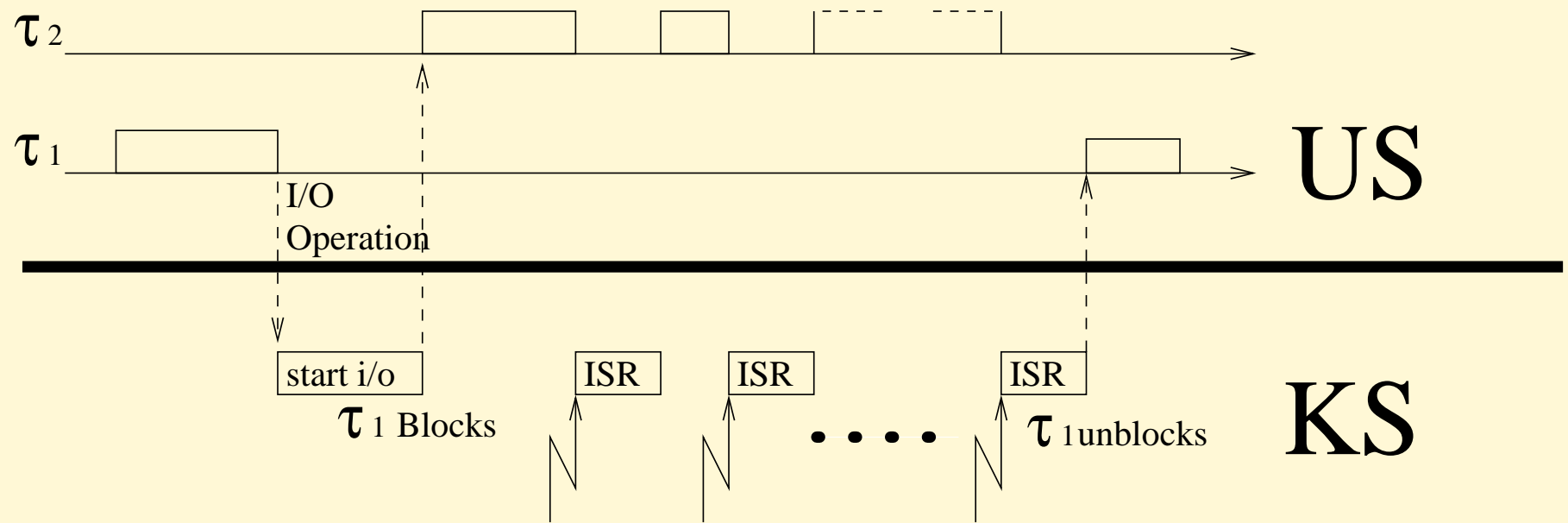
Interrupt

- User programs invoke the kernel; execution returns to user space while waiting for the device
 - The task that invoked the syscall blocks!
- An interrupt will notify the kernel when the “wait” phase is terminated
 - The interrupt handler will take care of performing the I/O operation
 - Many, frequent, short interruptions of unrelated user-space tasks!!!

Interrupt - 2

1. The user program raises a software input
2. Setup phase - in kernel: instruct the device to raise an input when it is ready for I/O
3. Wait - return to user space: block the invoking task, and schedule a new one (IRET)
4. Cleanup - in kernel: the interrupt fires → enter kernel, and perform the I/O operation
5. Return to phase 2, or unblock the task if the operation is terminated (IRET)

Programmed I/O Mode



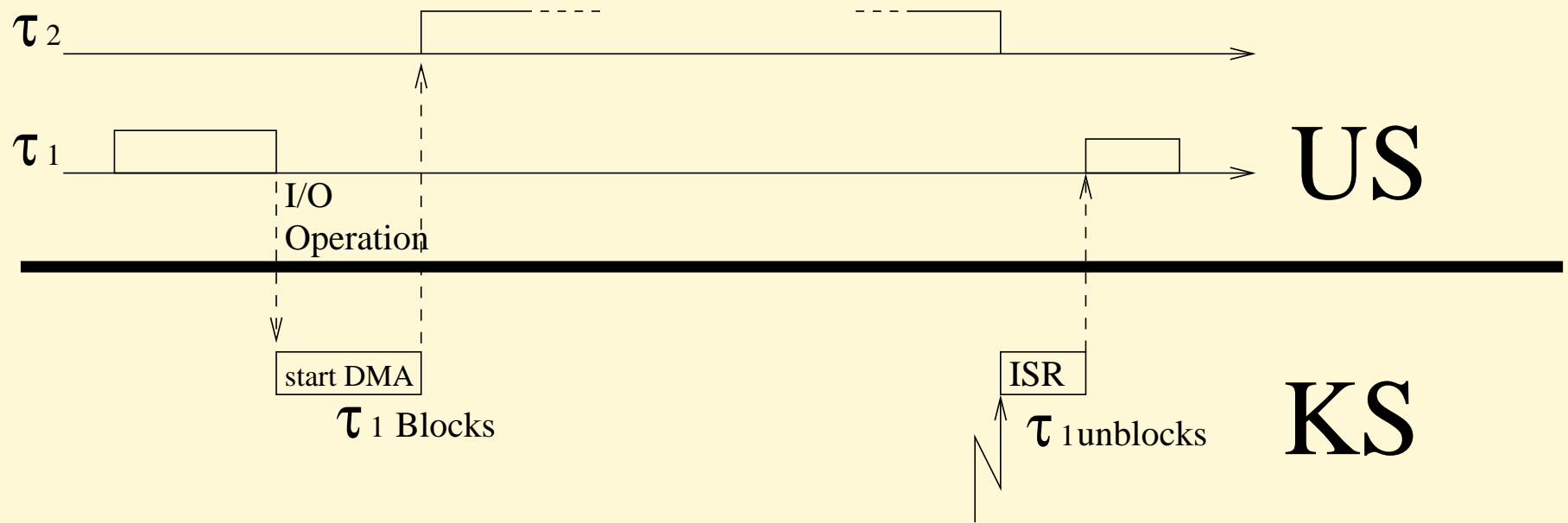
DMA / Bus Mastering

- User programs invoke the kernel; execution returns to user space while waiting for the device
 - The task that invoked the syscall blocks!
- I/O operations are not performed by the kernel on interrupt,
- Performed by a dedicated HW device
 - An interrupt is raised when the whole I/O operation is terminated

DMA / Bus Mastering - 2

1. The user program raises a software input
2. Setup phase - in kernel: instruct the DMA (or the Bus Mastering Device) to perform the I/O
3. Wait - return to user space: block the invoking task, and schedule a new one (IRET)
4. Cleanup - in kernel: the interrupt fires → the operation is terminated. Stop device and DMA
5. Unblock the task and invoke the scheduler (IRET)

DMA / Bus Mastering - 3



Example: Linux System Call

```
int close(int fd)
{
    long __res;

    __asm__ volatile ("int_$0x80"
        : "=a" (__res)
        : "0" (__NR_close), "b" ((long) (fd)));
    __syscall_return(type, __res);
}
```

- Don't be scared!
 - `__syscall_return()` is just converting a linux error code in `-1`, properly filling `errno`
- Linux uses a `_syscall1` macro to define it (see `asm/unistd.h`)

```
#define _syscall1(type, name, type1, arg1)
type name(type1 arg1) \
{ \
    ...
```

Kernel Side (arch/*/kernel/entry.S)

```
ENTRY(system_call)
pushl %eax # save orig_eax
SAVE_ALL
GET_THREAD_INFO(%ebp)
cmpl $(nr_syscalls), %eax
jae syscall_badsys
syscall_call:
call *sys_call_table(,%eax,4)
movl %eax,EAX(%esp) # store the return value
/* ... */
restore_all:
/* ... */
RESTORE_REGS
addl $4, %esp
1: iret
```

- `SAVE_ALL` pushes all the registers on the stack
- The `syscall` number is in the `eax` register (accumulator)
- After executing the `syscall`, the return value is in `eax`
→ must be put in the stack to pop it in `RESTORE_REGS`