Real Time Operating Systems

The Kernel

Luca Abeni

Real-Time Operating Systems

- Real-Time operating system (RTOS): OS providing support to Real-Time applications
- Operating System:
  - Set of computer programs
  - Interface between applications and hardware
  - Control the execution of application programs
  - Manage the hardware and software resources
  - Abstracts the physical machine, multiplexing it between executing tasks
- OS as...
  - A Service Provider for user programs (POSIX API...)
  - A Resource Manager

Operating System Services

- Services (Kernel Space):
  - Process / Thread Scheduling
  - Process Synchronisation, Inter-Process Communication (IPC)
  - I / O
  - Virtual Memory
- Provided to user tasks through an API
  - RT-POSIX interface

Task Scheduling

- The core part of the OS (the *kernel*) implements a virtual processor abstraction
  - A task set $\mathcal{T}$ composed by N tasks runs on M CPUs, with $M < N$
  - All tasks τ_i have the illusion to run in parallel
  - Temporal multiplexing between tasks
- Two core components:
  - The *scheduler* is responsible for deciding which task is executed
  - The *dispatcher* actually switches the CPU context to the context of the scheduled task (context switch)

Synchronization and IPC

- The kernel must also provide a mechanism for allowing tasks to communicate and synchronize
- Two possible programming paradigms:
  - Shared memory (threads)
    - The kernel must provide semaphores or / and mutexes + condition variables
    - Real-time resource sharing protocols (PI, HLP, NPP, ...) must be implemented
  - Message passing (processes)
    - Interaction models: pipeline, client / server, ...
    - The kernel must provide some IPC mechanism: pipes, message queues, mailboxes, Remote Procedure Calls (RPC), ...
    - Some real-time protocols can still be used

Real-Time Scheduling in Practice

- An adequate scheduling of system resources removes the need for over-engineering the system, and is necessary for providing a predictable QoS
- Algorithm + Implementation = Scheduling
- RT theory provides us with good algorithms...
- ...But which are the prerequisites for correctly implementing them?

Theoretical and Actual Scheduling

- Scheduler, IPC subsystem, and device drivers $\rightarrow$ must respect the theoretical model saw in previous lessons
  - Scheduling is simple: fixed priorities
  - IPC, HLP, or NPP are simple too...
  - But what about timers?
    - we already noticed some problems...
- Problem:
  - Are we sure that the scheduler is able to select a high-priority task as soon as it is ready?
  - And the dispatcher?

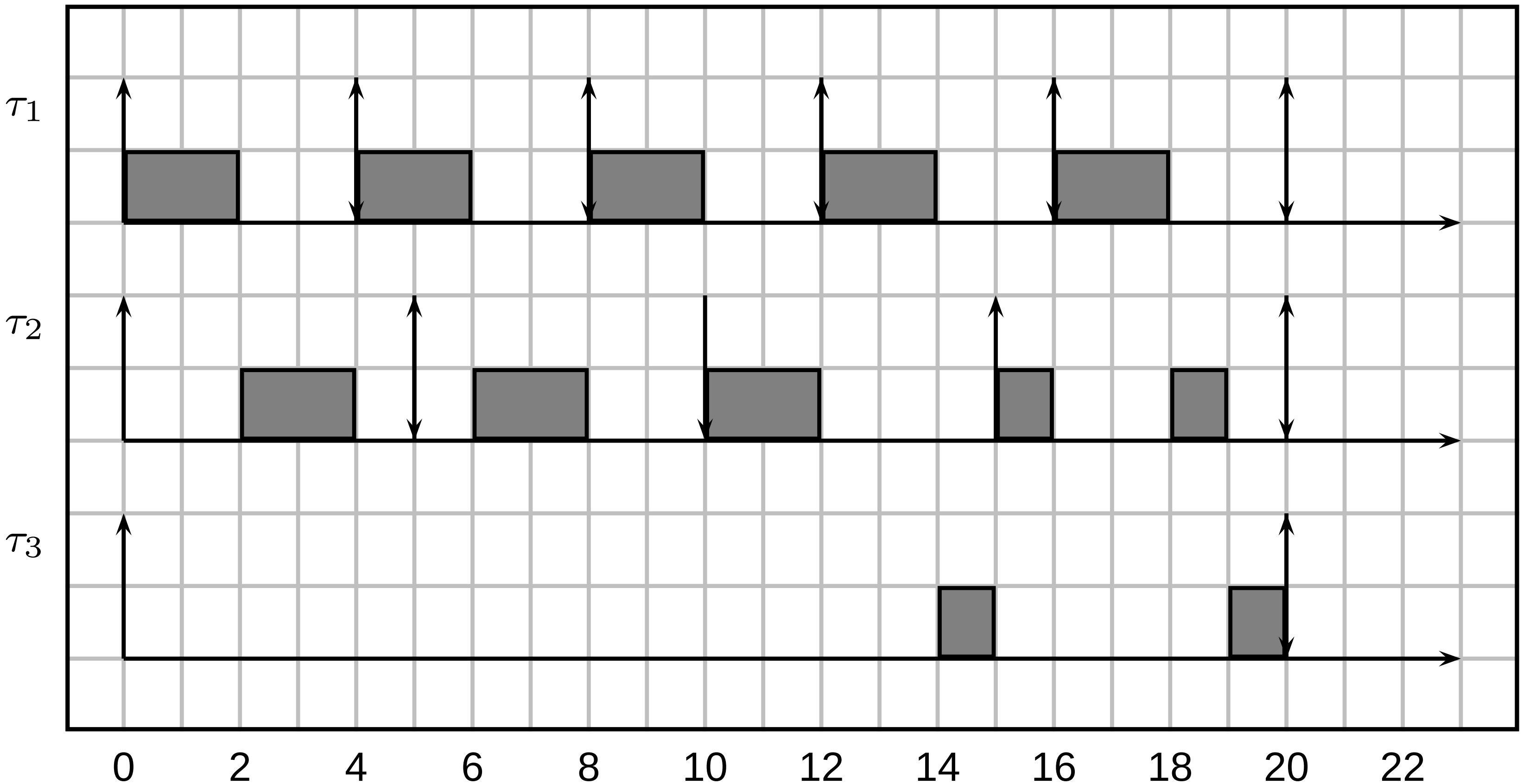
Periodic Task Example

- Consider a periodic task

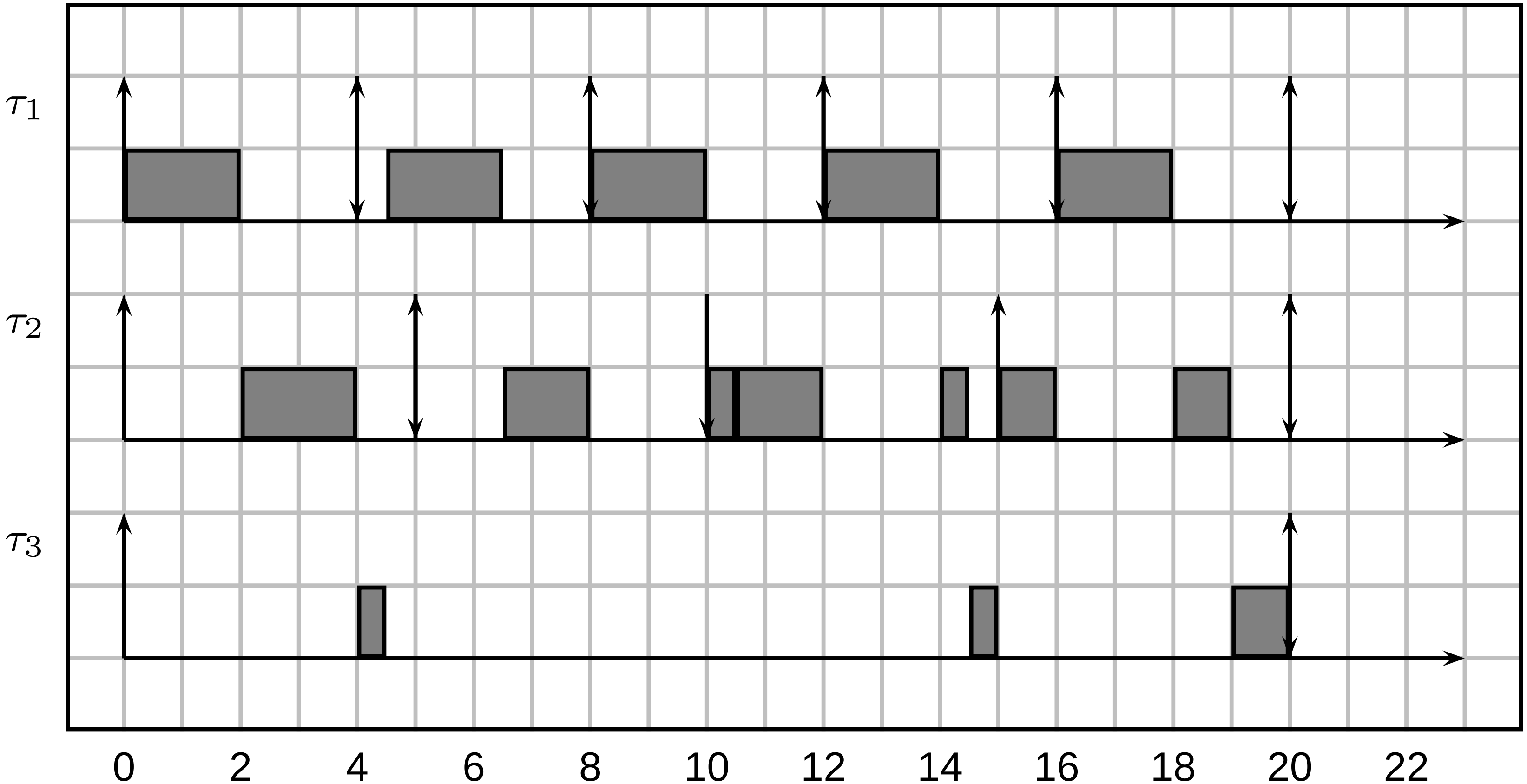
```
/* ... */
while(1) {
  /* Job body */
  clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &r, NULL);
  timespec_add_us(&r, period);
}
```

- The task expects to be executed at time r ($= r_0 + jT$)...
- ...But is sometimes delayed to $r_0 + jT + \delta$

Example - Theoretical Schedule



Example - Actual Schedule

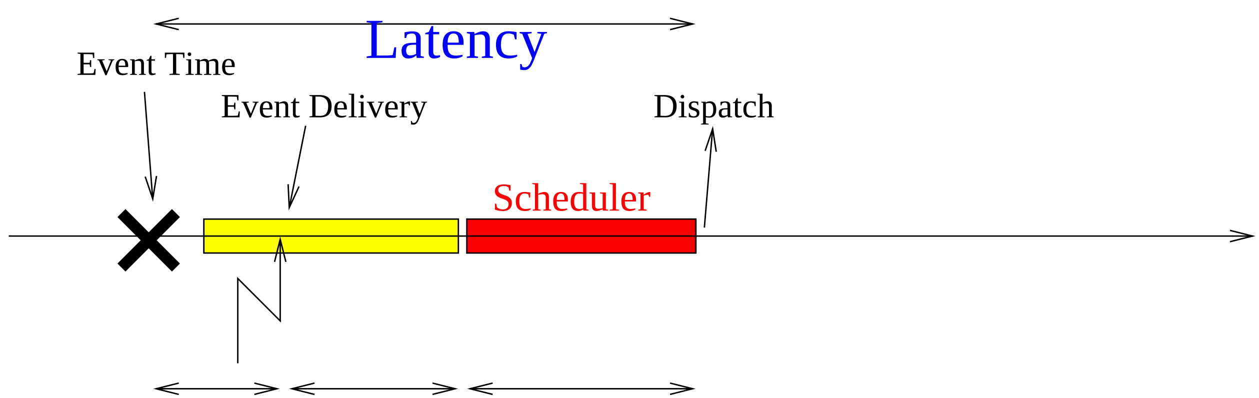


Kernel Latency

- The delay δ in scheduling a task is due to *kernel latency*
- Kernel latency can be modelled as a blocking time
  - $\sum_{k=1}^{N} \frac{C_k}{T_k} \leq U_{lub} \rightarrow \forall i,\ 1 \leq i \leq n, \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + \delta}{T_i} \leq U_{lub}$
  - $R_i = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h \rightarrow R_i = C_i + \delta + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$
  - $\exists 0 \leq t \leq D_i : W_i(0, t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h \leq t \rightarrow$
    $\exists 0 \leq t \leq D_i : W_i(0, t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h \leq t - \delta$

Kernel Latency

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

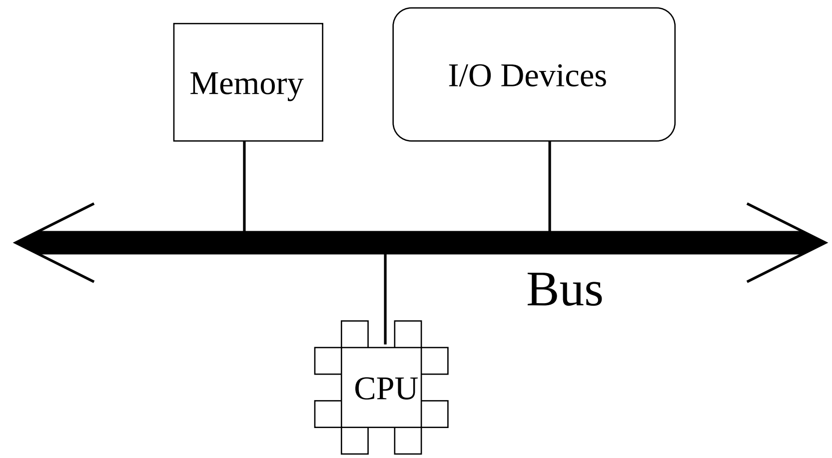


Theoretical Model vs Real Schedule

- 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...

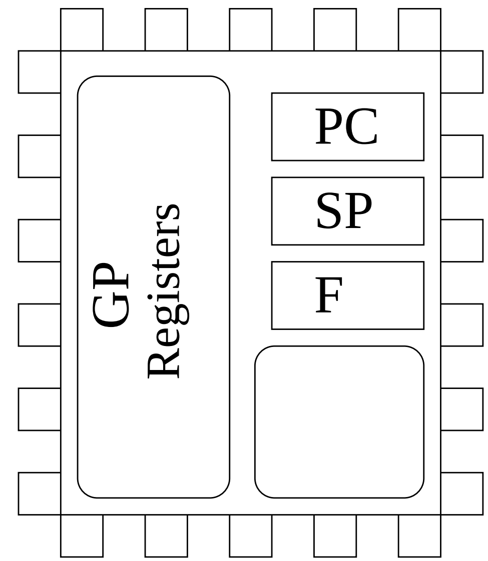
System Architecture

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



The CPU

- Some general-purpose registers
  - Can be accessed by all the programs
  - *data registers* or *address registers*
- Program Counter (PC) register (also known as Instruction Pointer)
- Stack Pointer (SP) register
- Flags register (also know as 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
- $\Rightarrow$ Need for “privileged” mode of execution (*Supervisor Mode*)
  - Regular registers vs “special” registers
  - Regular instructions vs privileged instructions
- User programs run at a low privilege level (*User Level*)
- Part of the OS (generally the *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
  - Segmented architecture $\rightarrow$ segment registers CS (code segment), DS (data segment), SS (stack segment), GS, FS
  - Various modes of operation: RM, PM, VM86, . . .

The Kernel

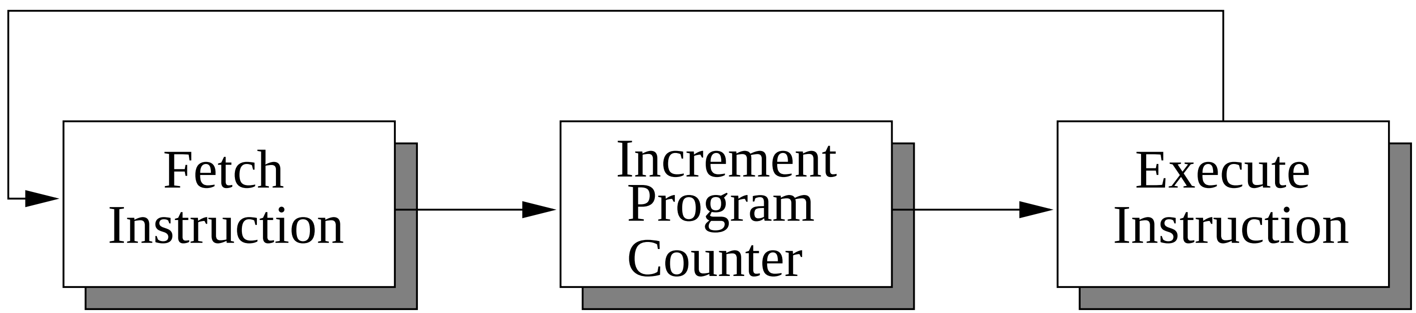
- Part of the OS which manages the hardware
- Runs with the CPU in *privileged mode* (high privilege level), or *Supervisor Mode*
  - We often say that the privilege level is the *Kernel Level* (KL), or execution is in *Kernel Space*
  - Regular programs run at *User Level* (UL), in *User Space*
- Some mechanism is needed for increasing the privilege level (from US to KS) in a controlled way
  - Interrupts (+ traps / hw execptions)
  - CPUs provide a way to switch to KL: software interrupts / instructions causing an 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 $\rightarrow$ save the user registers for restoring them later
- Execution returns to User Level through a “return from interrupt” instruction (`IRET` on x86)
  - Pop flags and PC from the stack
  - Eventually switch back to user stack

Simplified CPU Execution

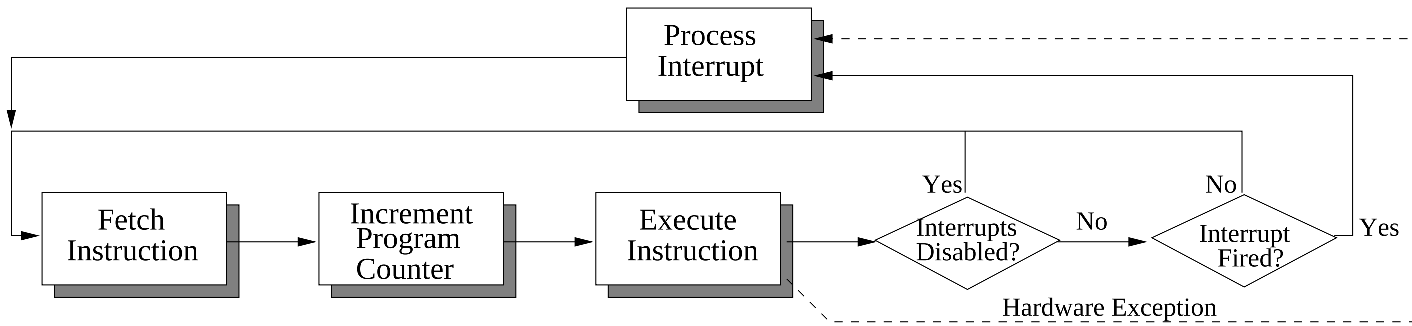
- To understand interrupts, consider simplified CPU execution first



- The CPU iteratively:
  - Fetch an instruction (address given by PC)
  - Increase the PC
  - Execute the instruction (might update the PC on jump...)

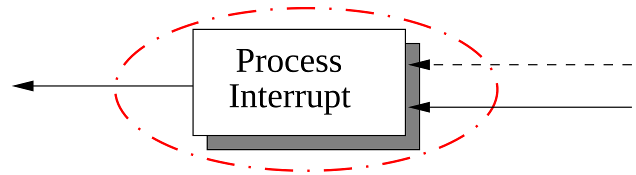
CPU Execution with Interrupts

- More realistic execution model



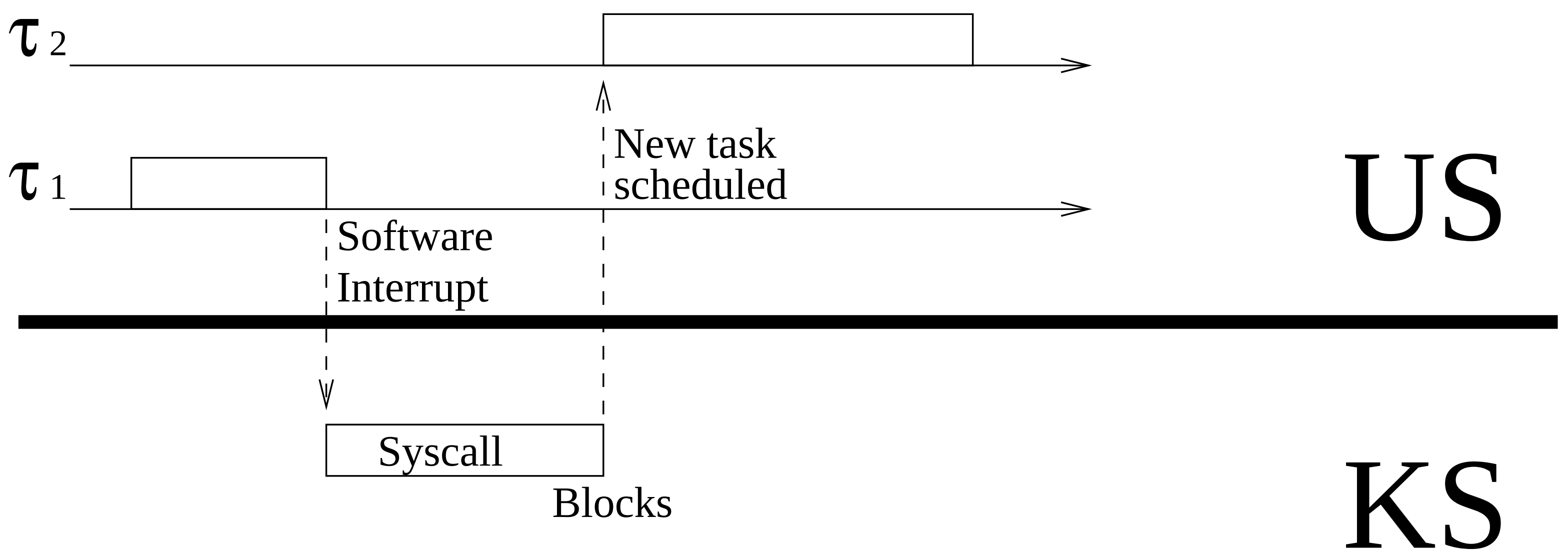
- Interrupt: cannot fire during the execution of an instruction
- Hardware exception: caused by the execution of an instruction
  - `trap`, `syscall`, `sc`, ...
  - I/O instructions at low privilege level
  - Page faults

Processing Interrupts



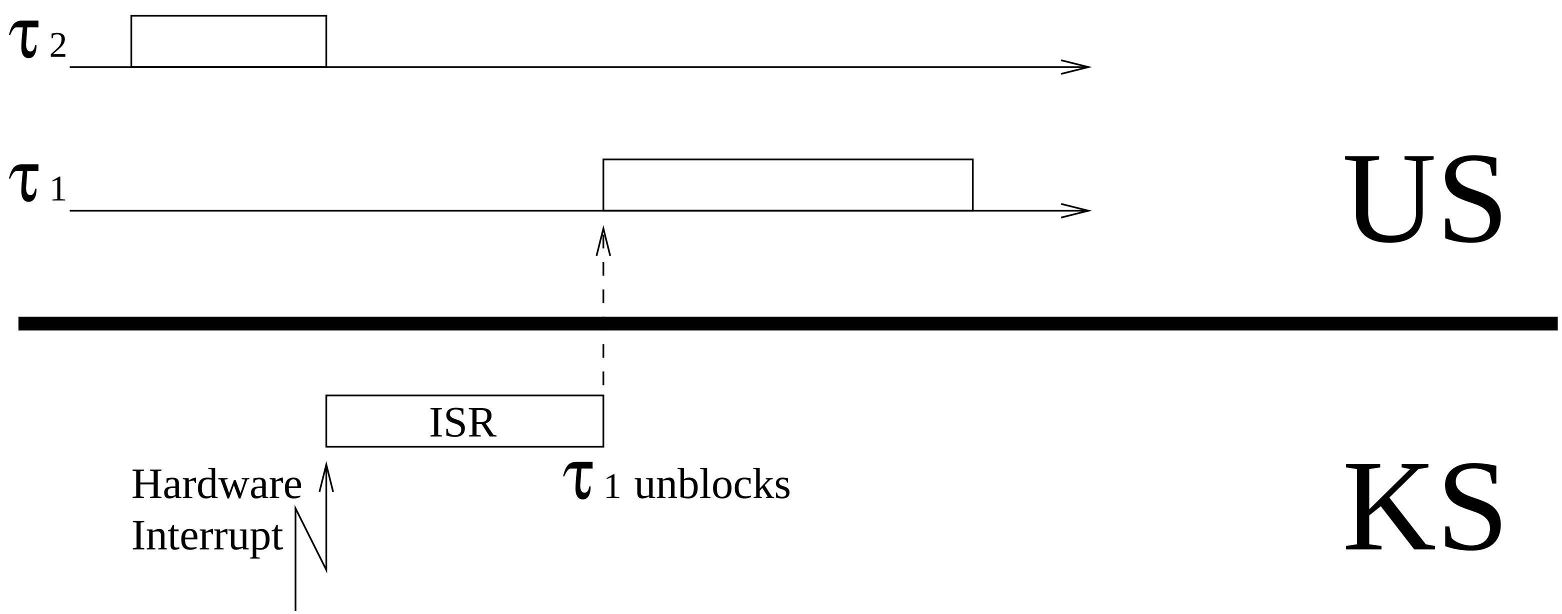
- *Interrupt table* $\rightarrow$ addresses of the handlers
  - Interrupt n fires $\Rightarrow$ after eventually switching to KS and pushing flags and PC on the stack
  - Read the address contained in the n^{th} entry of the interrupt table, and jump to it!
- Implemented in hardware or in software
  - x86 $\rightarrow$ **I**nterrupt **D**escription **T**able composed by interrupt gates. The CPU automatically jumps to the n^{th} interrupt gate
  - Other CPUs jump to a fixed address $\rightarrow$ a software demultiplexer reads the interrupt table

Software Interrupt - System Call



1. Task τ_1 executes and invokes a system call (by issuing a software interrupt)
2. Execution passes from US to KS (the stack is changed, PC & flags are pushed, privilege level is increased)
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 task τ_2 is executing, an hardware interrupt arrives
2. Execution passes from US to KS (the stack is changed, PC & flags are pushed, privilege level is increased)
3. The proper **I**nterrupt **S**ervice **R**outine executes
4. The ISR can unblock $\tau_1 \rightarrow$ when execution returns to US, τ_1 is scheduled

Summing up...

- The execution flow enters the kernel for two reasons:
  - Reacting to an event “coming from up” (a syscall)
  - Reacting to an event “coming from down” (an hardware interrupt from a device)
- The kernel executes in the context of the interrupted task
- 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 use a syscall for accessing the device
- The operation if performed in 3 phases
  1. Setup: prepare the device for the I/O operation
  2. Wait: wait for the device to terminate the operation
  3. Cleanup: complete the operation
- Various way to perform the operation: polling, PIO, DMA, ...

Polling

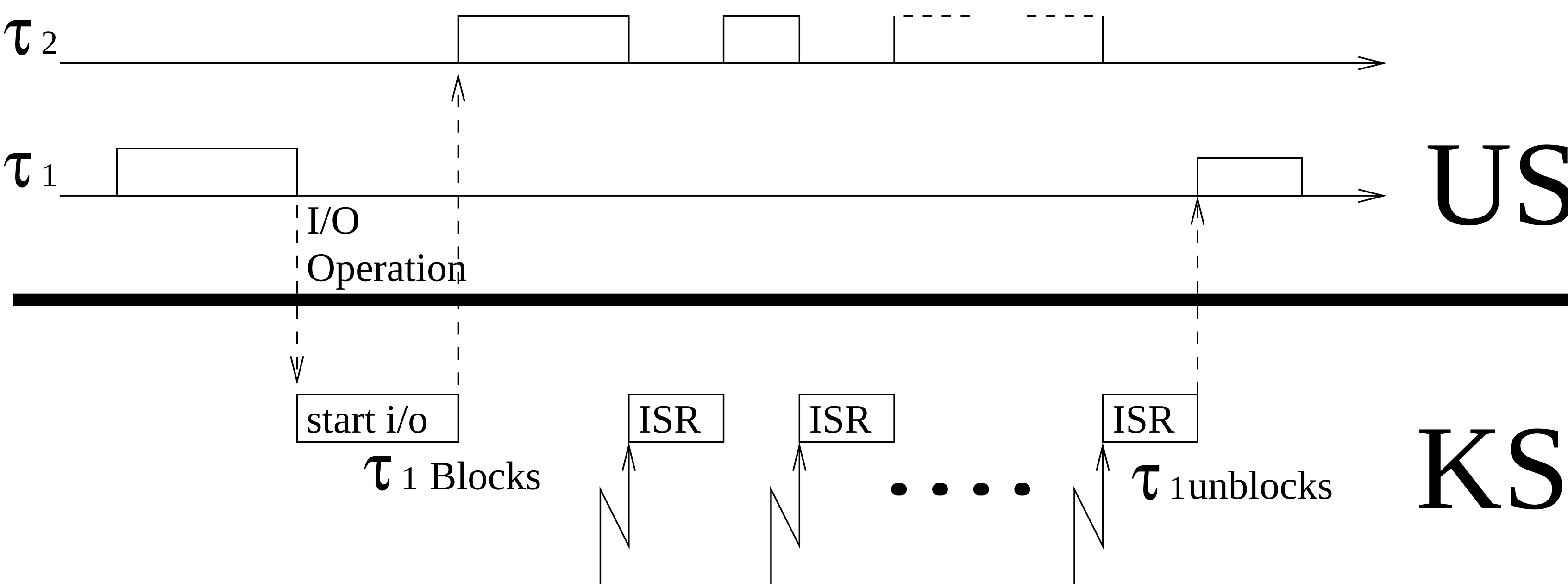
- The user program invokes the kernel, and execution remains in kernel space until the operation is terminated
- The kernel cyclically reads (polls) an interface status register to check if the operation is terminated

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

- The user program invokes the kernel, but execution returns to user space (the process blocks) while waiting for the device
- An interrupt will notify the kernel that phase 2 is terminated
  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 $\rightarrow$ enter kernel, and perform the I/O operation
  5. Return to phase 2, or unblock the task if the operation is terminated (IRET)

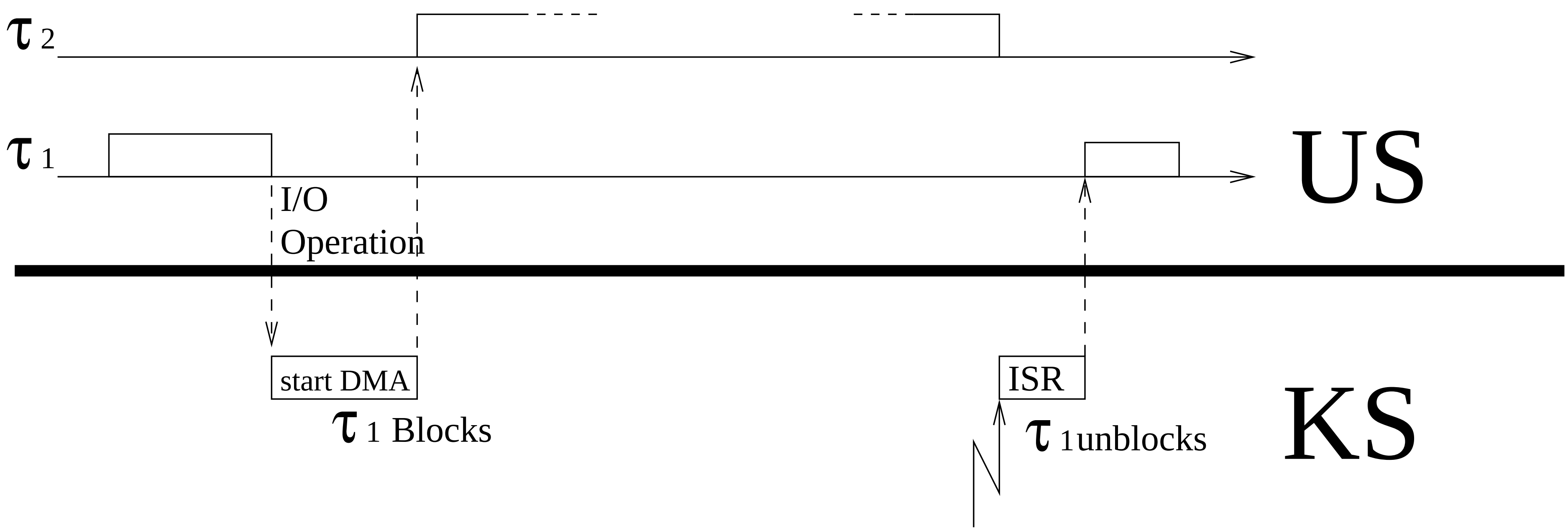
Programmed I/O Mode



DMA / Bus Mastering

- The user program invokes the kernel, but execution returns to user space (the process blocks) while waiting for the device
- I/O operations are not performed by the kernel on interrupt, but by a dedicated HW device. An interrupt is raised when the whole I/O operation is terminated
  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 $\rightarrow$ the operation is terminated. Stop device and DMA
  5. Unblock the task and invoke the scheduler (IRET)

DMA / Bus Mastering - 2



Example: Linux System Call

```c
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`)

```c
#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