The Kernel

Luca Abeni
`luca.abeni@santannapisa.it`

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 execeptions)
  - Instructions causing a hardware exception

Interrupts and Hardware Exceptions

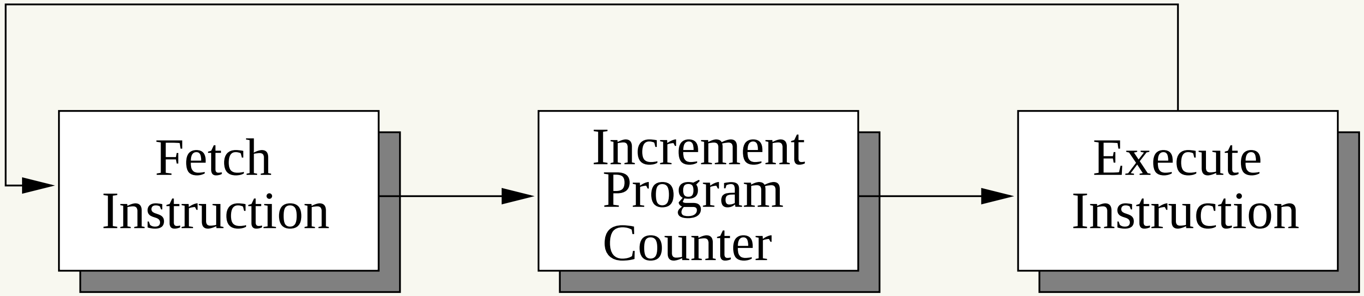
- Switch the CPU from User Level to Supervisor Mode
  - Enter the kernel
  - Can be used to implement *system calls*
- A partial Context Switch is performed
  - Flags and PC are pushed on the stack
  - If processor is executing at User Level, switch to Kernel Level, and eventually switch to a *kernel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

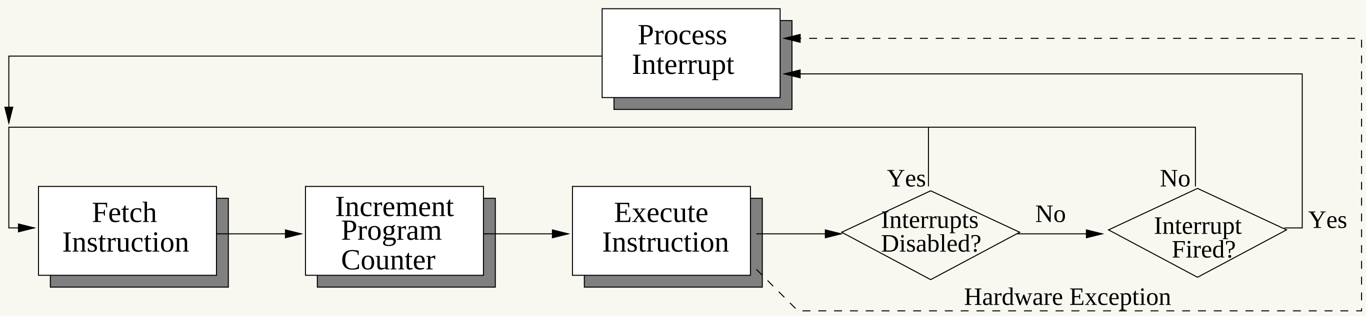
- To understand interrupts, consider simplified CPU execution first
  - Simplification respect to the fetch/decode/load/execute/save cycle



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

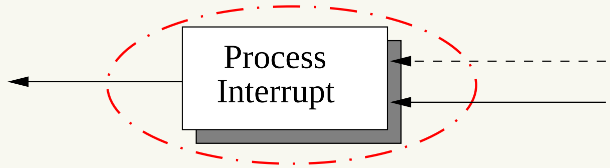
CPU Execution with Interrupts

- 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

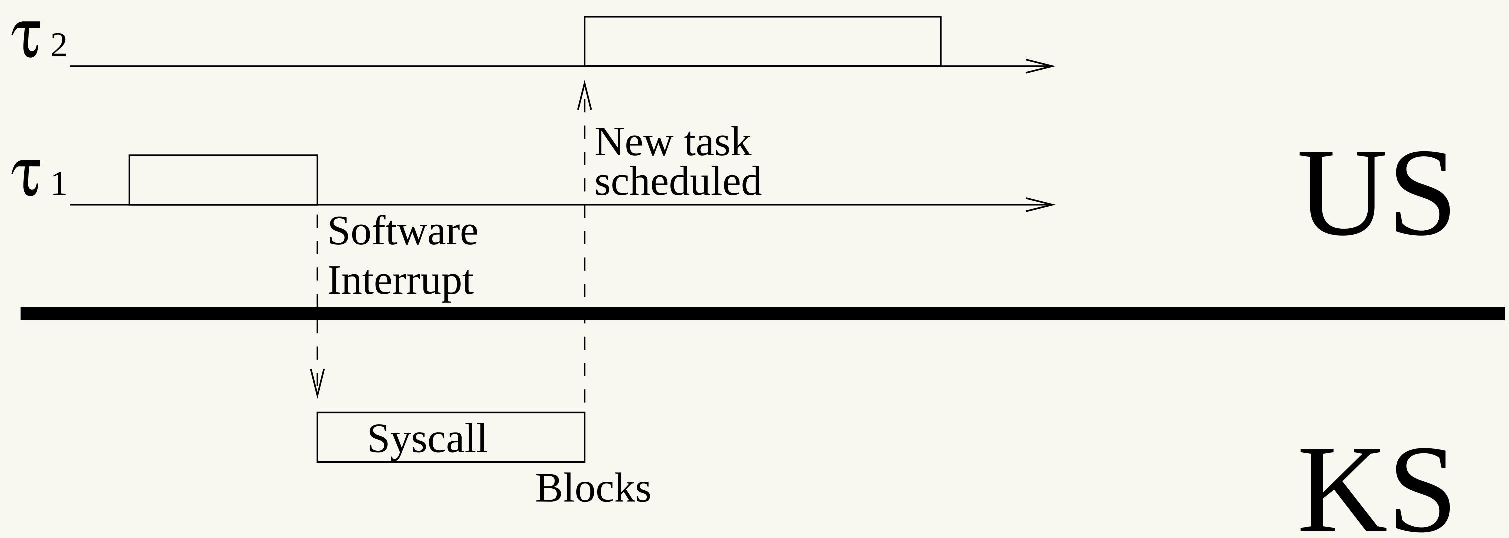


- *Interrupt table* $\rightarrow$ addresses of the handlers
  - Interrupt n fires $\Rightarrow$ after eventually switching to KS and pushing flags and PC on the stack
  - Read the address contained in the n^{th} entry of the interrupt table, and jump to it!

Interrupt Tables

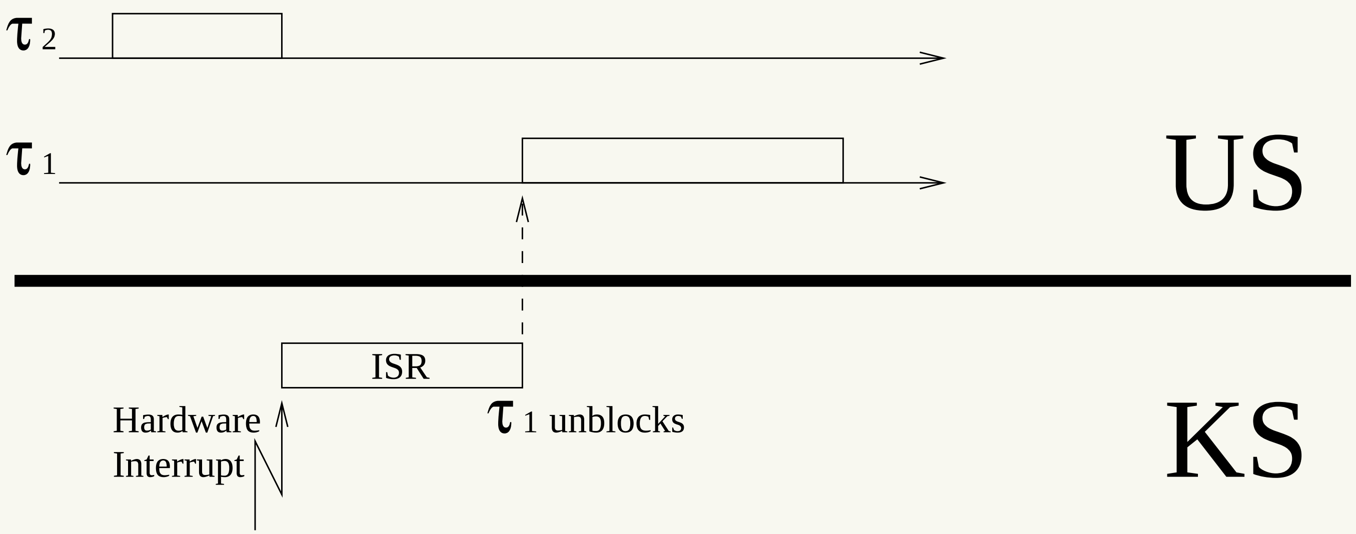
- Implemented in hardware or in software
  - x86 $\rightarrow$ **I**nterrupt **D**escription **T**able composed of interrupt gates. The CPU automatically jumps to the n^{th} interrupt gate
  - Other CPUs jump to a fixed address $\rightarrow$ a software demultiplexer reads the interrupt table

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 if 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

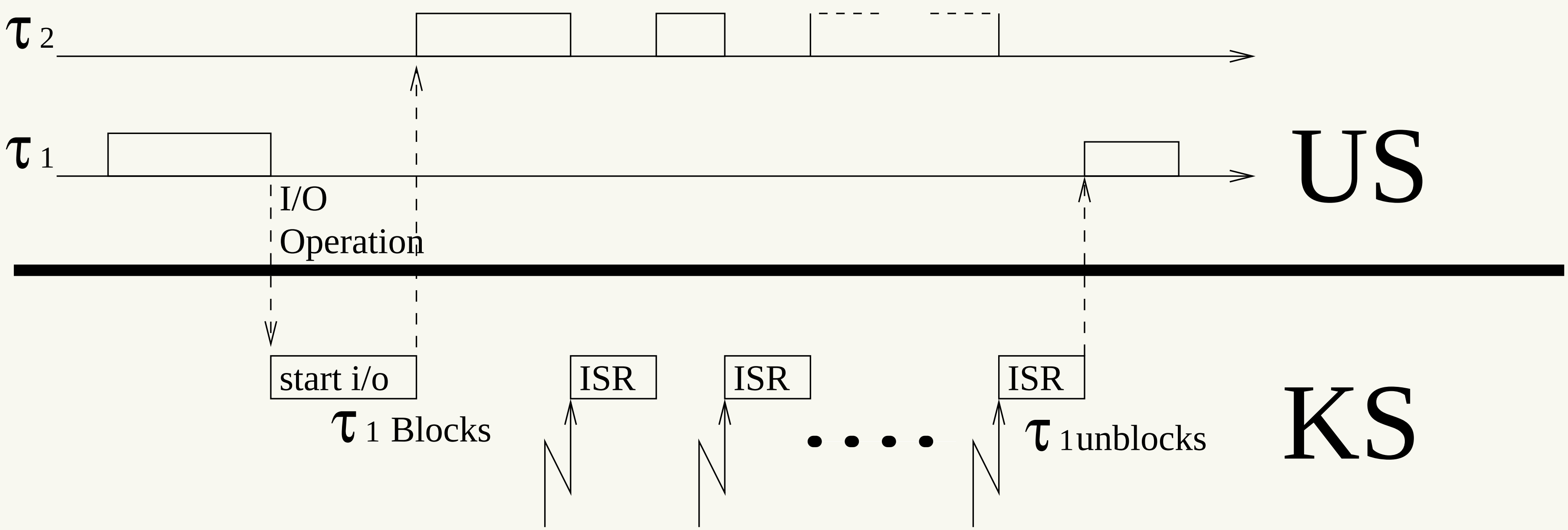
Interrupt

- 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 $\rightarrow$ 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



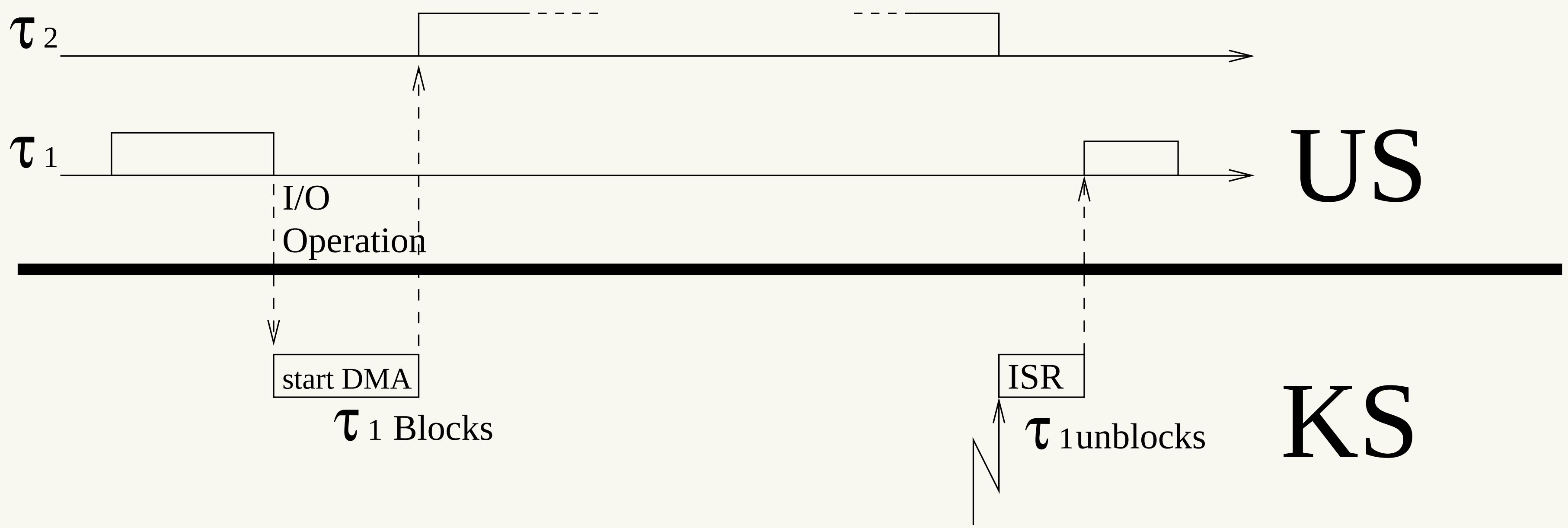
DMA / Bus Mastering

- 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

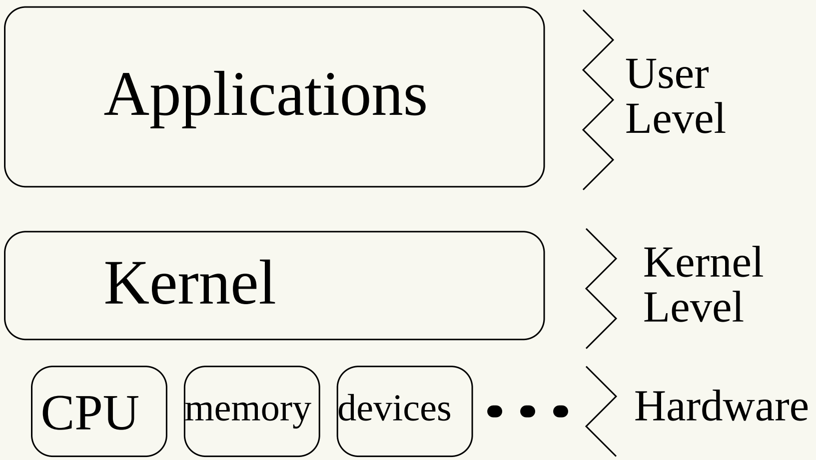
1. The user program raises a software input
2. Setup phase - in kernel: instruct the DMA (or the Bus Mastering Device) to perform the I/O
3. Wait - return to user space: block the invoking task, and schedule a new one (IRET)
4. Cleanup - in kernel: the interrupt fires $\rightarrow$ the operation is terminated.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

DMA / Bus Mastering - 3



Invoking the Kernel

- Kernel $\rightarrow$ part of an OS that interacts with the hardware
  - Runs with CPU in privileged mode
  - User Level $\Leftrightarrow$ Kernel Level switch through special CPU instructions (`INT` for Intel x86)
- User Level applications
  - Run with the CPU in non-privileged mode
  - invoke *system calls* or IPCs



System Libraries

- Applications generally don't invoke system calls directly
- They generally use *system libraries* (like glibc), which
  - Provide a more advanced user interface (example: `fopen()` vs `open()`)
  - Hide the US $\Leftrightarrow$ KS switches
  - Provide some kind of stable ABI (application binary interface)
- Example: let's see how system calls are converted in regular library calls

System Library Example

- Standard C library: exports some functions...
- ...That are just converted in system calls! (example: `getpid()`)
- Let's see how this works...
  - Some Assembly is needed

```
syscall:
   pushl %ebp
   pushl %edi
   pushl %esi
   pushl %ebx

/* arguments in registers */
   movl  44(%esp),%ebp
   movl  40(%esp),%edi
   /*...*/
   int $0x80

   popl %ebx
   /*...*/
```

```
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,PT_EAX(%esp) # store the ret val
syscall_exit:
  /*...*/
```

Static vs Shared Libraries - 1

- Libraries can be *static* or *dynamic*
  - `<libname>.a` vs `<libname>.so`
- Static libraries (`.a`)
  - Collections of object files (`.o`)
  - Application linked to a static library $\Rightarrow$ the needed objects are included into the executable
  - Only needed to compile the application

Static vs Shared Libraries - 2

- Dynamic libraries (`.so`, shared objects)
  - Are not included in the executable
  - Application linked to a dynamic library $\Rightarrow$ only the library symbols names are written in the executable
  - Actual linking is performed at loading time
  - `.so` files are needed to execute the application
- Linking static libraries produces larger executables...
- ...But these executables are “self contained”