

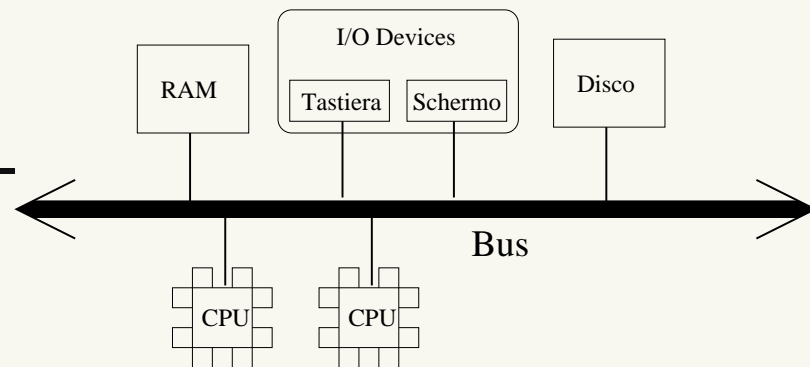
# *The Kernel*

Luca Abeni

luca.abeni@santannapisa.it

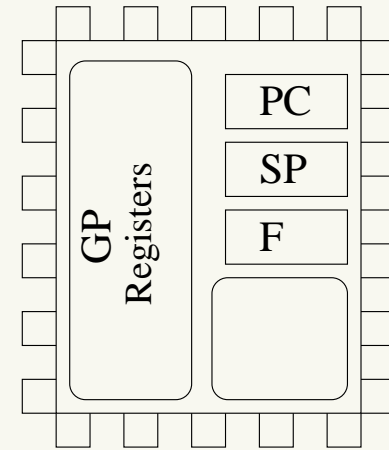
# System Architecture

- 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
  - *data registers* or *address registers*



- 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

# 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

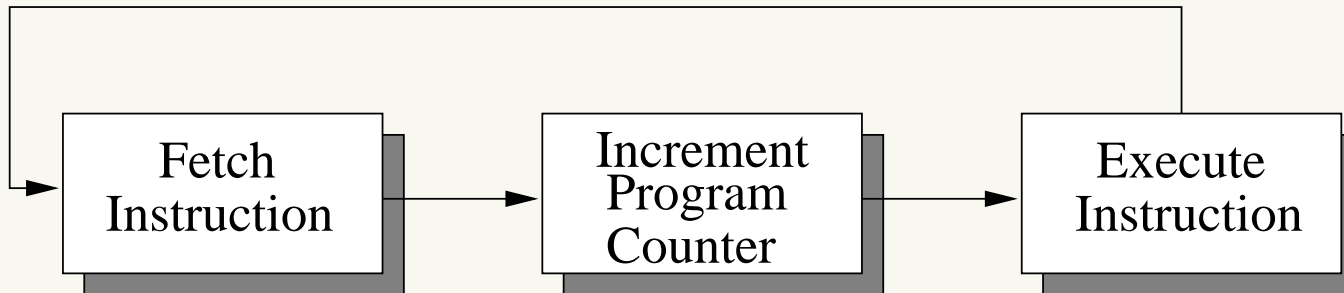


# Back to User Space

- 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

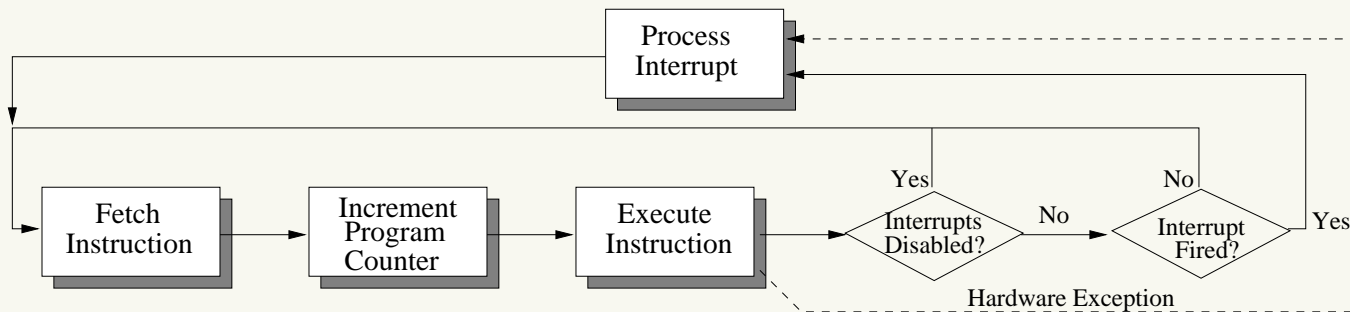
- To understand interrupts, consider simplified CPU execution first



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

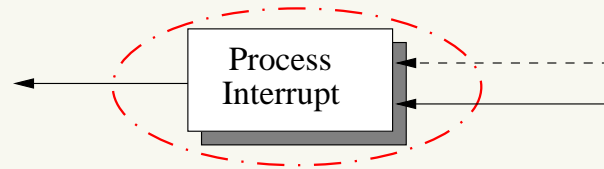
# CPU Execution with Interrupts

- 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

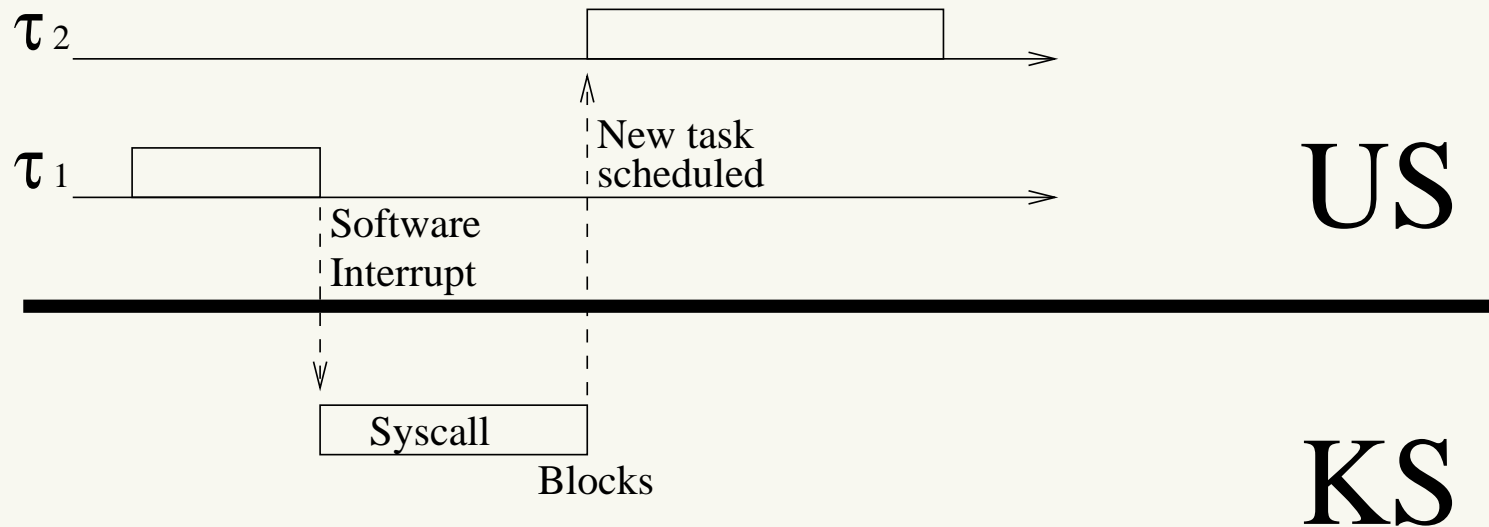


- *Interrupt table* → addresses of the handlers
  - Interrupt  $n$  fires  $\Rightarrow$  after eventually switching to KS and pushing flags and PC on the stack
  - Read the address contained in the  $n^{th}$  entry of the interrupt table, and jump to it!

# Interrupt Tables

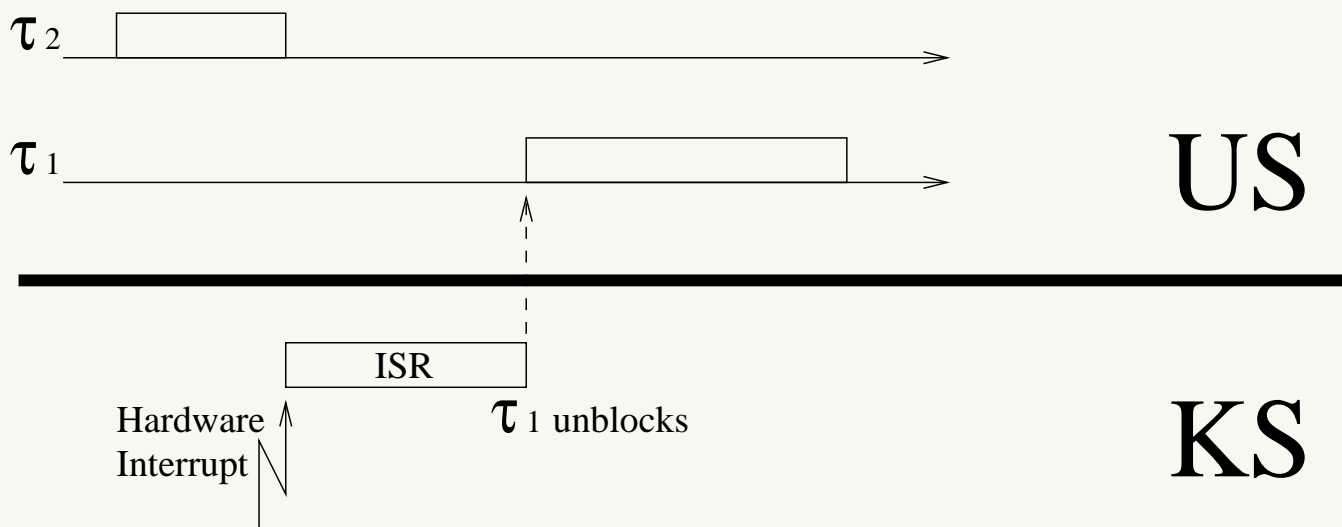
- Implemented in hardware or in software
  - x86 → **I**nterrupt **D**escription **T**able composed by interrupt gates. The CPU automatically jumps to the  $n^{th}$  interrupt gate
  - Other CPUs jump to a fixed address → a software demultiplexer reads the interrupt table

# Software Interrupt - System Call



1. Task  $\tau_1$  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.  $\tau_1$  blocks  $\rightarrow$  back to US, and  $\tau_2$  is scheduled

# Hardware Interrupt



1. While  $\tau_2$  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  $\tau_1$   $\rightarrow$  when execution returns to US,  $\tau_1$  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

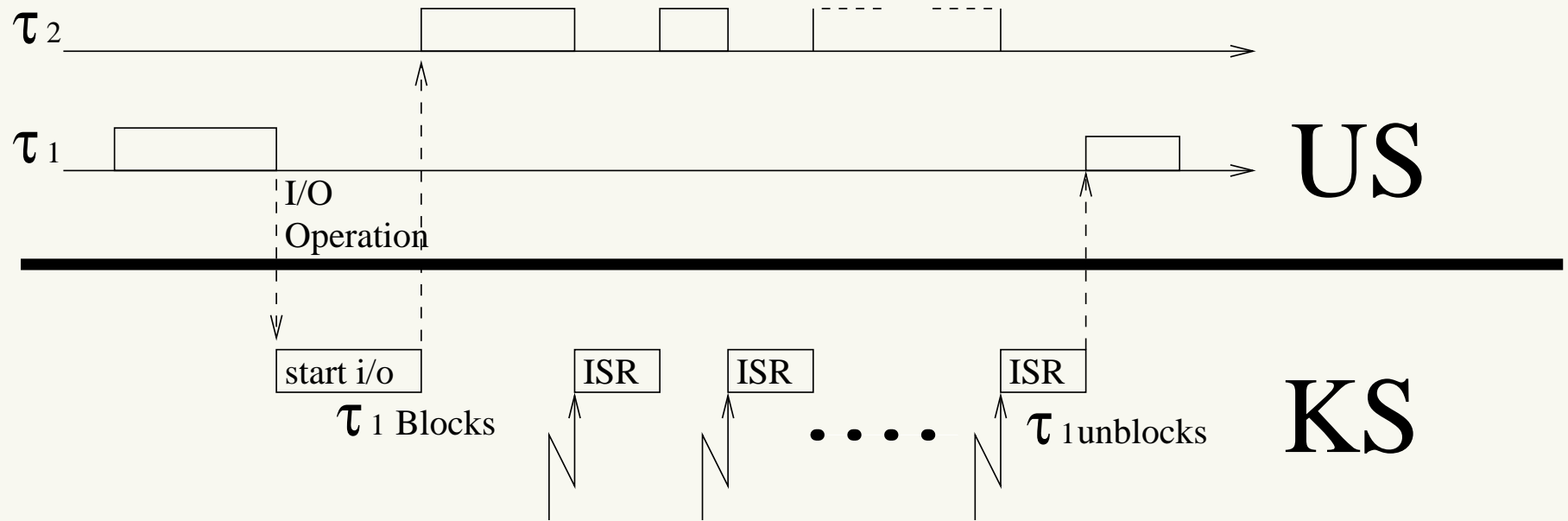
# Interrupt

- 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

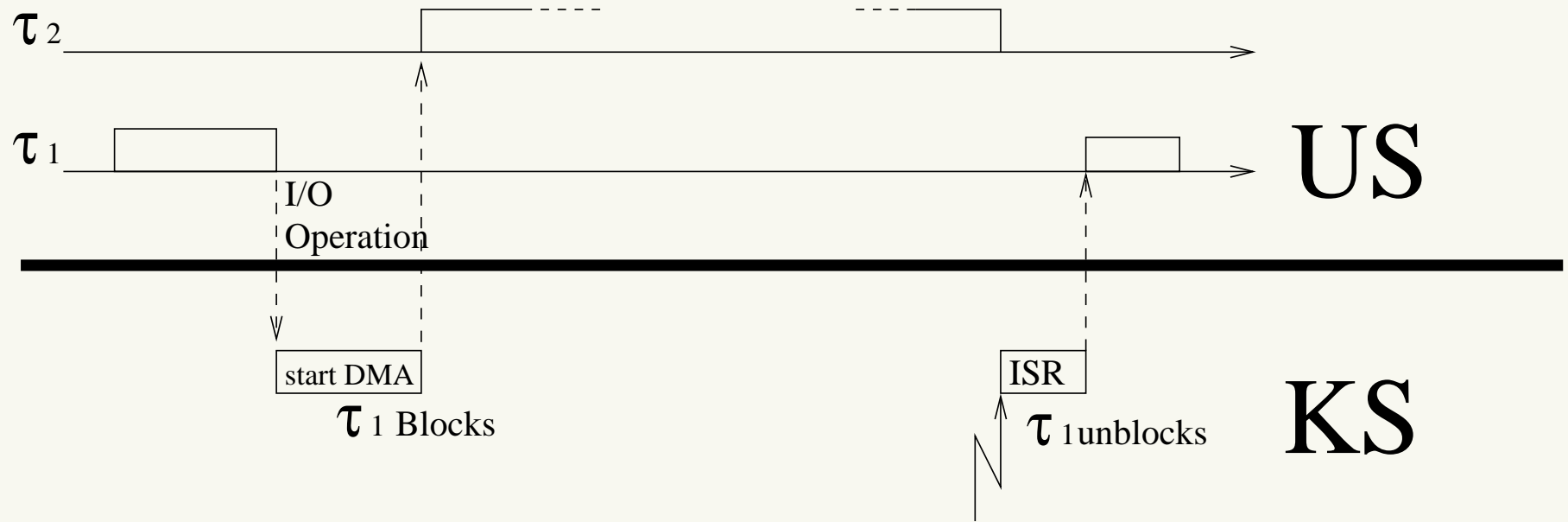
- User programs invoke the kernel; execution returns to user space while waiting for the device
  - The task that invoked the syscall blocks!
- I/O operations are not performed by the kernel on interrupt,
- Performed by a dedicated HW device
  - An interrupt is raised when the whole I/O operation is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



## DMA / Bus Mastering - 2

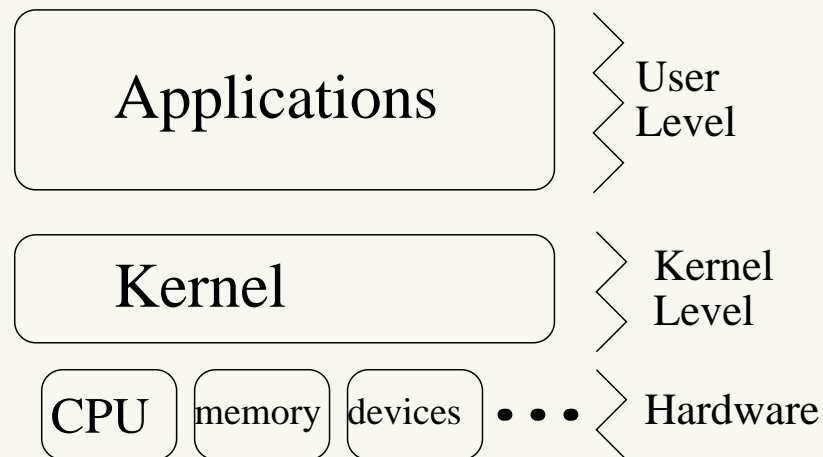
1. The user program raises a software input
2. Setup phase - in kernel: instruct the DMA (or the Bus Mastering Device) to perform the I/O
3. Wait - return to user space: block the invoking task, and schedule a new one (IRET)
4. Cleanup - in kernel: the interrupt fires → the operation is terminated. Stop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

# DMA / Bus Mastering - 3



# Invoking the Kernel

- Kernel → 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”

# Monolithic Kernels

- Traditional Unix-like structure
- Protection: distinction between Kernel (running in KS) and User Applications (running in US)
- The kernel behaves as a single-threaded program
  - One single execution flow in KS at each time
  - Simplify consistency of internal kernel structures
- Execution enters the kernel in two ways:
  - Coming from upside (system calls)
  - Coming from below (hardware interrupts)



# Single-Threaded Kernels

- Only one single execution flow (thread) can execute in the kernel
  - It is not possible to execute more than 1 system call at time
    - Non-preemptable system calls
    - In SMP systems, syscalls are critical sections (execute in mutual exclusion)
  - Interrupt handlers execute in the context of the interrupted task

# Bottom Halves

- Interrupt handlers split in two parts
  - Short and fast ISR
  - “Soft IRQ handler”
- Soft IRQ handler: *deferred* handler
  - Traditionally known as Bottom Half (BH)
  - AKA Deferred Procedure Call - DPC - in Windows
  - Linux: distinction between “traditional” BHs and Soft IRQ handlers

# Synchronizing System Calls and BHs

- Synchronization with ISRs by disabling interrupts
- Synchronization with BHs: is almost automatic
  - BHs execute atomically (a BH cannot interrupt another BH)
  - BHs execute at the end of the system call, before invoking the scheduler for returning to US
- Easy synchronization, but large non-preemptable sections!
  - Achieved by reducing the kernel parallelism
  - Can be bad for real-time

# Latency in Single-Threaded Kernels

- Kernels working in this way are often called *non-preemptable kernels*
- $L^{np}$  is upper-bounded by the maximum amount of time spent in KS
  - Maximum system call length
  - Maximum amount of time spent serving interrupts

# Evolution of the Monolithic Structure

- Monolithic kernels are single-threaded: how to run them on multiprocessor?
  - The kernel is a critical section: Big Kernel Lock protecting every system call
  - This solution does not scale well: a more fine-grained locking is needed!
- Tasks cannot block on these locks → not mutexes, but *spinlocks*!
  - Remember? When the CS is busy, a mutex **blocks**, a spinlock **spins**!
  - Busy waiting... Not that great idea...

# Removing the Big Kernel Lock

- Big Kernel Lock → huge critical section **for everyone**
  - Bad for real-time...
  - ...But also bad for throughput!
- Let's split it in multiple locks...
- Fine-grained locking allows more execution flows in the kernel simultaneously
  - More parallelism in the kernel...
  - ...But tasks executing in kernel mode are still non-preemptable

# Preemptable Kernels

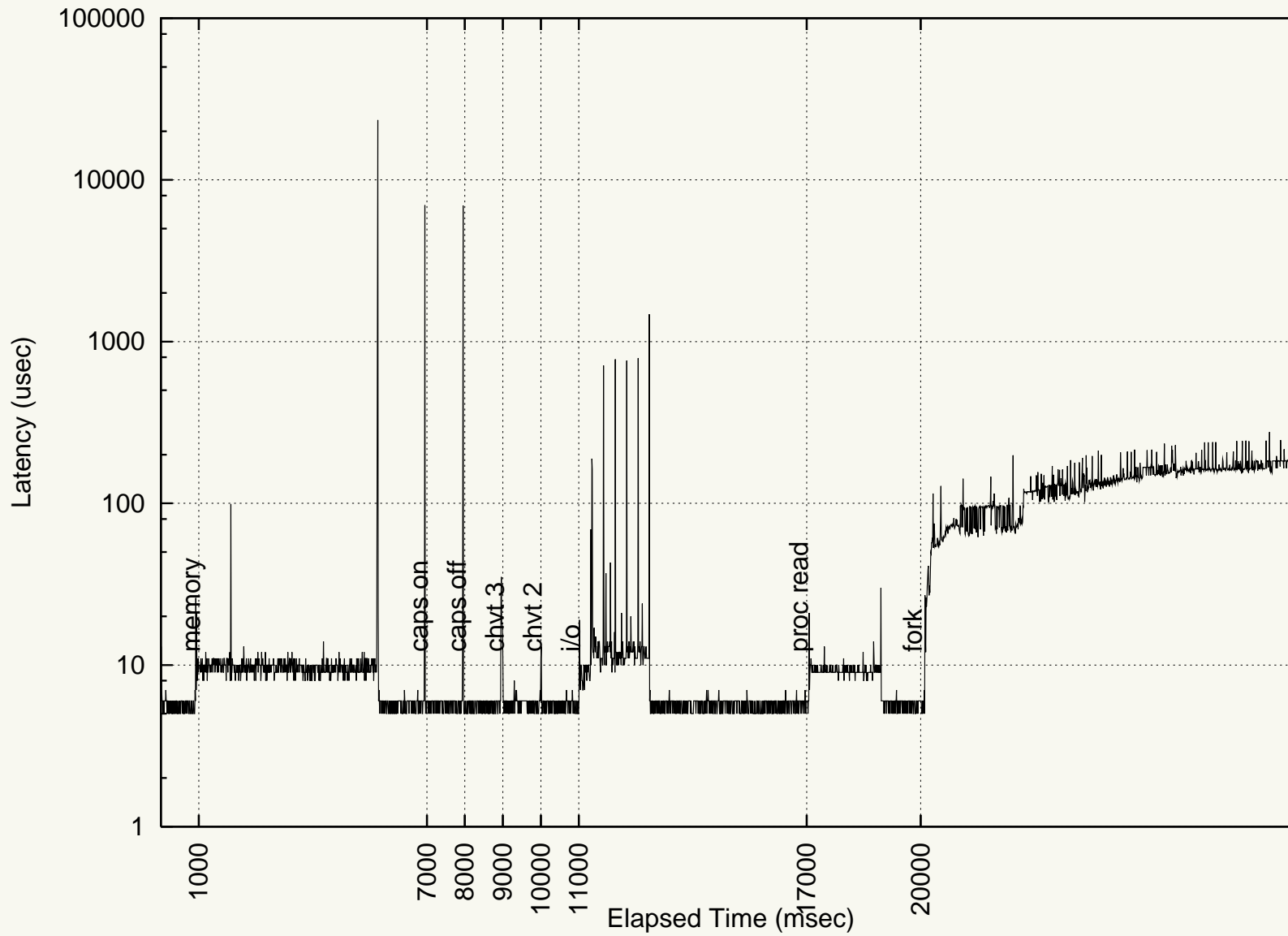
- Multithreaded kernel
  - Fine-grained critical sections inside the kernel
  - Kernel code is still non-preemptable
- Idea: When the kernel is not in critical section, preemptions can occur
  - Check for preemptions when exiting kernel's critical sections

# Linux Kernel Preemptability

- Check for preemption when exiting a kernel critical section
  - Implemented by modifying spinlocks
  - Preemption counter: increased when locking, decreased when unlocking
  - When preemption counter == 0, check for preemption
- In a preemptable kernel,  $L^{np}$  is upper bounded by the maximum size of a kernel critical section
- Critical section == non-preemptable... **This is NPP!!!**



# Latency in a Preemptable Kernel



# NPP Drawbacks

- Preemptable Kernel: use NPP for kernel critical sections
- NPP is known to have issues
  - Low-priority tasks with large critical sections can affect the schedulability of high-priority tasks not using resources!
  - In this context: low-priority (or NRT) tasks invoking long system calls can compromise the schedulability of high priority real-time tasks (even if they do not use long system calls!)
- Can we do better???

# Doing Better than NPP

- Possible alternatives: **HLP** and **PI**
- HLP: easy to implement, but requires to know which resources the tasks will use
  - Possible to avoid high latencies on tasks not using the “long critical sections”, but...
  - ...Those tasks must be identified somehow!
- PI: does not impose restrictions or require a-priori knowledge of the tasks behaviour, but requires more changes to the kernel!

# Using HLP

- Simple idea: distinction between RT tasks (**do not use the kernel!**) and NRT tasks (can use the kernel)
- How the hell can we **execute a task without using the OS kernel???**
- Some “lower level RT-kernel” is needed
  - Running **below** the kernel!
  - Two possibilities:  $\mu$ kernels or dual-kernel systems