Real Time Operating Systems

The Non-Preemptable Sections Latency

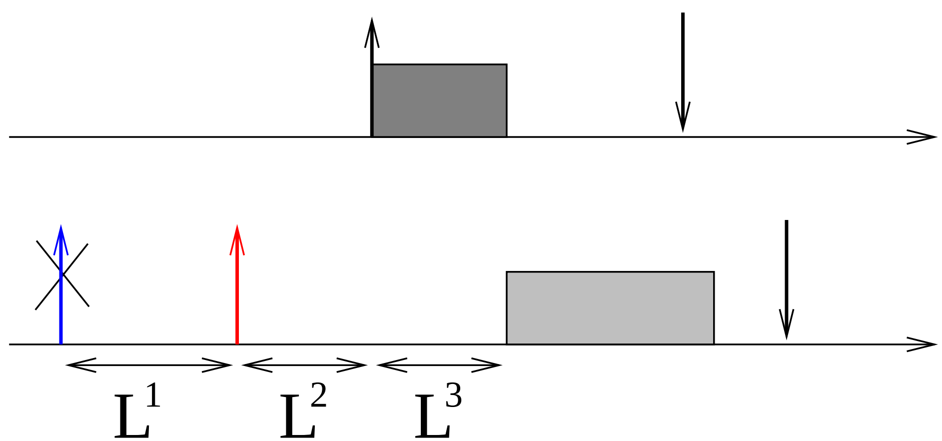
Luca Abeni

Latency

- Latency: measure of the difference between the theoretical and actual schedule
  - Task τ expects to be scheduled at time t ...
  - ... but is scheduled at time t'
  - $\Rightarrow$ Latency $L = t' - t$
- The latency L can be modelled as a blocking time $\Rightarrow$ affects the guarantee test
- If L is too high, only few task sets result to be schedulable
  - The latency must be *bounded*: $\exists L^{max} : L < L^{max}$
  - The latency bound L^{max} cannot be too high

Sources of Latency

- A task τ_i is a stream of jobs $J_{i,j}$ arriving at time $r_{i,j}$
- Job $J_{i,j}$ is scheduled at time $t' > r_{i,j}$
  - $t' - r_{i,j}$ is given by the sum of various components:
    1. $J_{i,j}$'s arrival is signalled at time $r_{i,j} + L^1$
    2. Such event is served at time $r_{i,j} + L^1 + L^2$
    3. $J_{i,j}$ is actually scheduled at $r_{i,j} + L^1 + L^2 + L^3$



Analysis of the Various Sources

- $L = L^1 + L^2 + L^3$
- L^3 is the *scheduler latency*
  - Interference from higher priority tasks
  - Already accounted by the guarantee tests $\rightarrow$ let's not consider it
- L^2 is the *non-preemptable section latency*, called L^{np}
  - Due to non-preemptable sections in the kernel, which delays the response to hardware interrupts
  - It is composed by various parts: *interrupt disabling*, *bottom halves delaying*, $\ldots$
- L^1 is due to the delayed interrupt generation

Interrupt Generation Latency

- Hardware interrupts are generated by external devices
- Sometimes, a device must generate an interrupt at time $t \ldots$
- $\ldots$ but actually generates it at time $t' = t + L^{int}$
- L^{int} is the *Interrupt Generation Latency*
  - It is due to hardware issues
  - It is *generally* small compared to L^{np}
  - Exception: if the device is a timer device, the interrupt generation latency can be quite high
    - *Timer Resolution Latency* L^{timer}
- The timer resolution latency L^{timer} can often be much larger than the non-preemptable section latency L^{np}

The Timer Resolution Latency

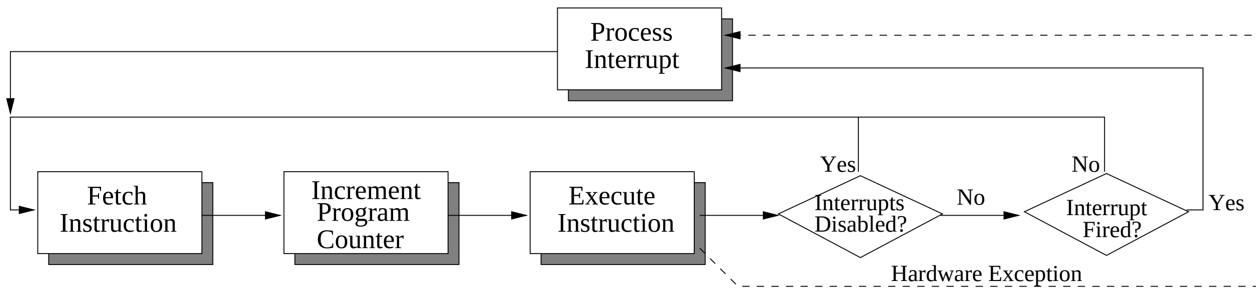
- Kernel timers are generally implemented by using a hardware device that produces periodic interrupts
- Periodic timer interrupt $\rightarrow$ tick
- Example: periodic task (`setitimer()`, Posix timers, `clock_nanosleep()`, $\ldots$) τ_i with period T_i
- At the end of each job, τ_i sleeps for the next activation
- Activations are triggered by the periodic interrupt
  - Periodic tick interrupt, with period T^{tick}
  - Every T^{tick}, the kernel checks if the task must be woken up
  - If T_i is not multiple of T^{tick}, τ_i experiences a timer resolution latency

Non-Preemptable Section Latency

- The *non-preemptable section latency* L^{np} is given by the sum of different components
  1. Interrupt disabling
  2. Delayed interrupt service
  3. Delayed scheduler invocation
- The first two are mechanisms used by the kernel to guarantee the consistency of internal structures
- The third mechanism is sometimes used to reduce the number of preemptions and increase the system throughput

Disabling Interrupts

- Remember? Before checking if an interrupt fired, the CPU checks if interrupts are enabled...



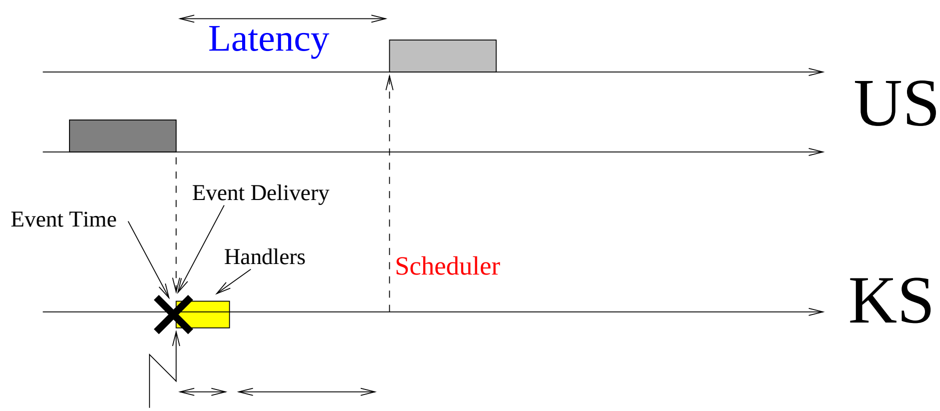
- Every CPU has some *protected* instructions (`STI/CLI` on x86) for enabling/disabling interrupts
  - Only the kernel (or code running in KS) can enable/disable interrupts
  - Interrupts disabled for a time $T^{cli} \rightarrow L^{np} \geq T^{cli}$
- Interrupt disabling is used to enforce mutual exclusion between sections of the kernel and ISRs

Delayed Interrupt Service

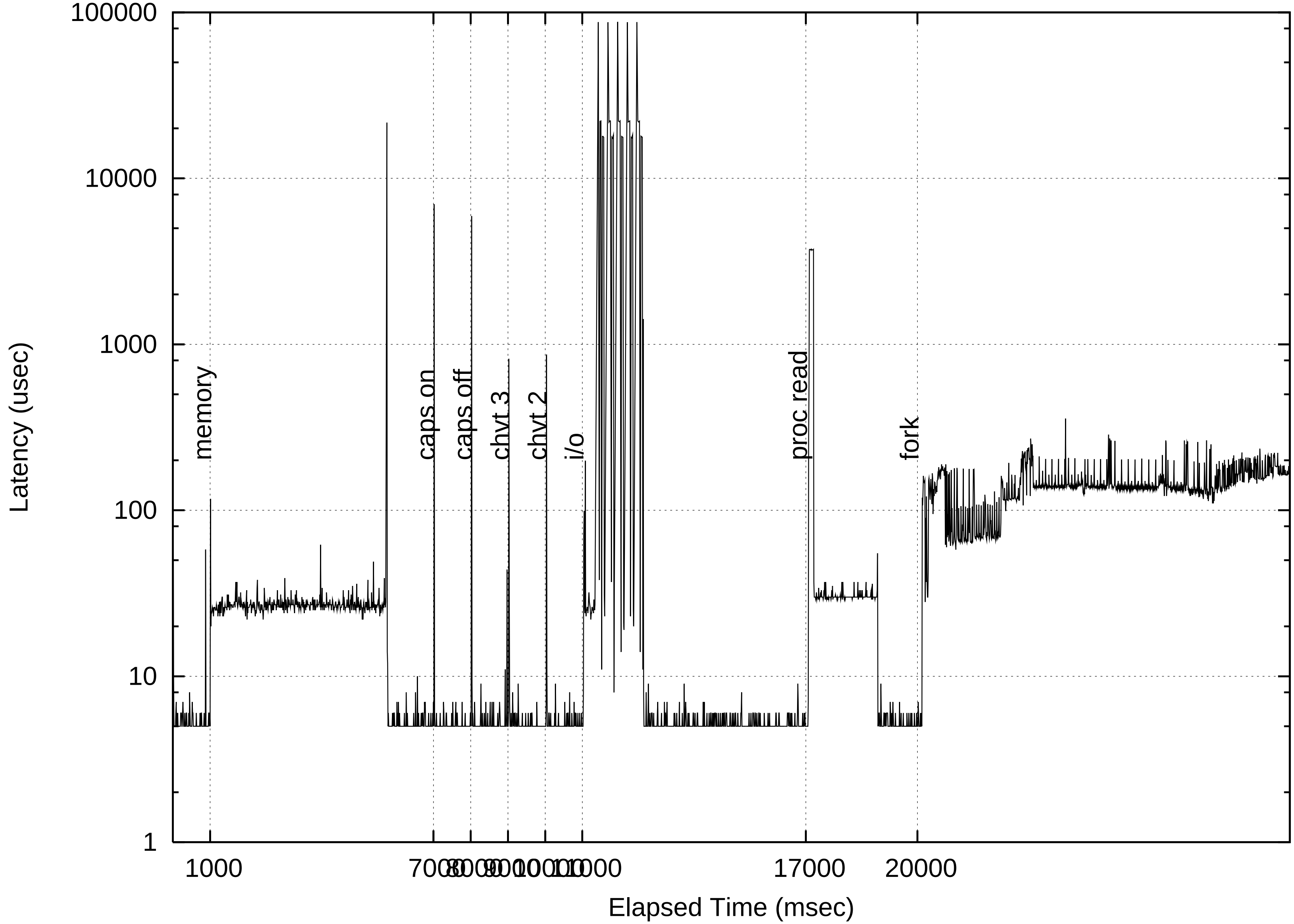
- When the interrupt fire, the ISR is ran, but the kernel can delay interrupt service some more...
  - ISRs are generally small, and do only few things
  - An ISR can set some kind of *software flag*, to notify that the interrupt fired
  - Later, the kernel can check such flag and run a larger (and more complex) interrupt handler
- Advantages of “larger interrupt handlers”:
  - They can re-enable interrupts
  - Enabling/Disabling such handlers is simpler/cheaper
- Disadvantages:
  - Interrupt response latency is increased: $L^{np} >> T^{cli}$
  - “larger interrupt handlers” are often non-preemptable

Deferred Scheduling

- Scheduler: invoked only when returning from KS to US
- For efficiency reasons, the kernel might want to return to user tasks only after performing a lot of activities
  - Try to reduce the number of KS $\leftrightarrow$ US switches
  - Reduce the number of context switches
  - Throughput vs low latency: opposite requirements
- So, maybe the ISR runs at the correct time, the delayed interrupt handler is ran im-
mediately, but the scheduler is invoked after some time...



Latency in the Standard Kernel



Summing Up

- L^{np} depends on some different factors
- In general, no hw reasons $\rightarrow$ it almost entirely depends on the *kernel structure*
  - Non-preemptable section latency is generally the result of the strategy used by the kernel for ensuring mutual exclusion on its internal data structures
  - To analyze / reduce L^{np}, we need to understand such strategies
  - Different kernels, based on different structures, work in different ways
- Some of the problems:
  - Interrupt Handling (Device Drivers)
  - Management of the parallelism

Data Structures Consistency

- Hardware interrupt: *breaks* the regular execution flow
  - If the CPU is executing in US, switch to KS
  - If execution is already in KS, possible problems
- Example:
  1. The kernel is updating a linked list
  2. IRQ While the list is in an inconsistent state
  3. Jump to the ISR, that needs to access the list...
- The kernel must *disable the interrupts* while updating the list!
- Similar interrupt disabling is also used in spinlocks and mutex implementations...

Real-Time Executives

- Executive: Library code that can be directly linked to applications
- Implements functionalities generally provided by kernels
- Generally, no distinction between US and KS
  - No CPU privileged mode, or application executes in privileged mode
  - “kernel” functionalities are invoked by direct function call
  - Applications can execute privileged instructions
- Advantages:
  - Simple, small, low overhead
  - Only the needed code is linked in the final image

Real-Time Executives - 2

- Disadvantages:
  - No protection
  - Applications can even disable interrupts $\rightarrow L^{np}$ risks to be unpredictable
- Examples:
  - RTEMS `http://www.rtems.org`
  - SHaRK `http://shark.sssup.it`
- Consistency of the internal structures is generally ensured by disabling interrupts: L^{np} is bounded by the maximum amount of time interrupts are disabled
- Generally used only when memory footprint is important, or when the CPU does not provide a privileged mode

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
  - Only one single execution flow runs in KS at each time
  - This greatly simplifies ensuring the consistency of internal kernel structures
- Execution enters the kernel in two ways:
  - Coming from up (system calls)
  - Coming from down (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
- Interrupt handlers split in two parts
  - Short and fast ISR
  - *Deferred* handler: Bottom Half (BH) (AKA Deferred Procedure Call - DPC - in Windows)

Synchronizing System Calls and BHs

- Synchronization with ISRs by disabling interrupts
- Synchronization with BHs is almost automatic: BHs execute at the end of the system call, before invoking the scheduler for returning to US
- BHs execute atomically (a BH cannot interrupt another BH)
- 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 then 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 $\rightarrow$ not mutexes, but *spinlocks*!
- 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

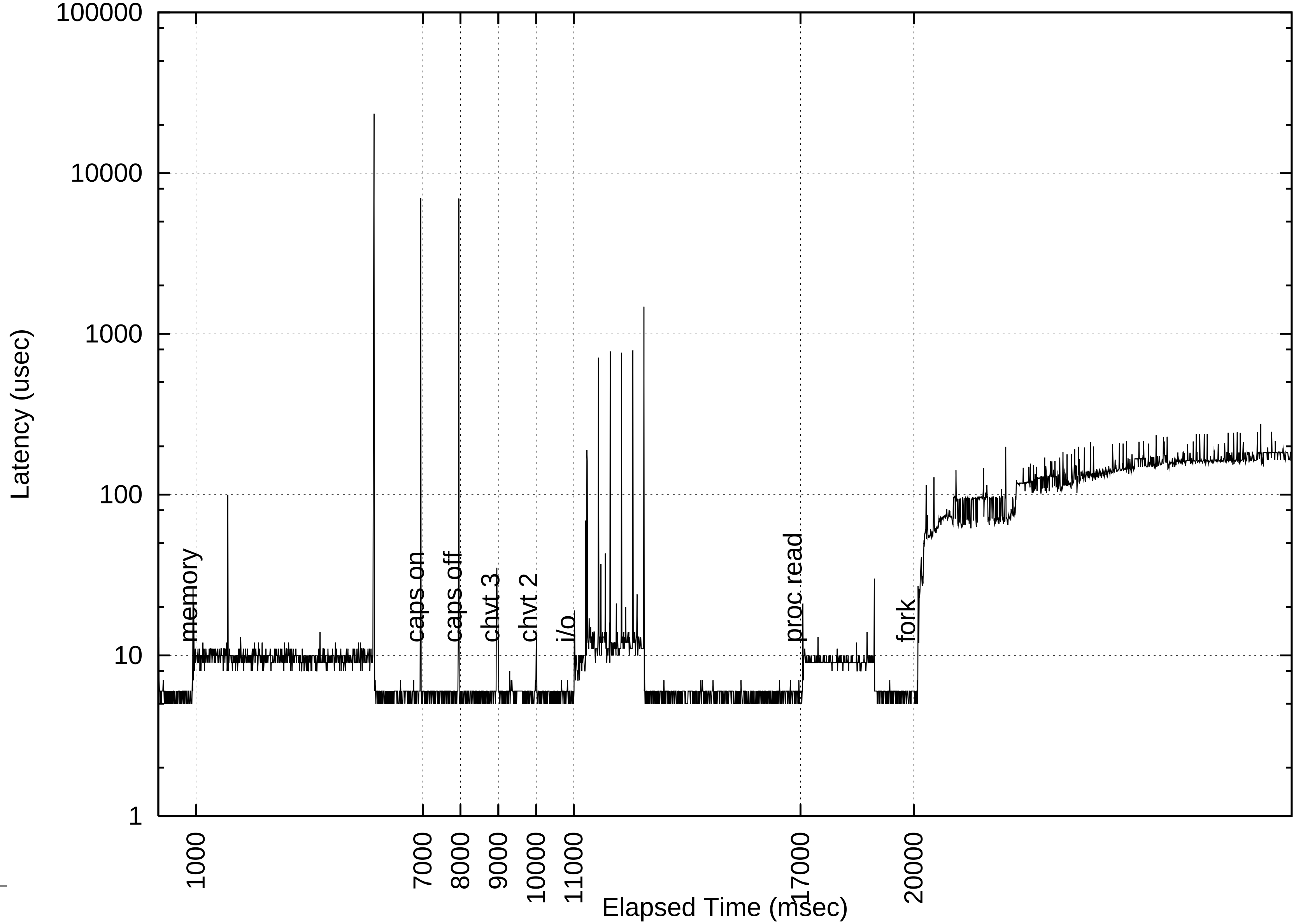
Spinlocks

- Spinlock: non-blocking synchronization object, similar to mutex
- Behave as a mutex, but tasks do not block on it
- A task trying to acquire an already locked spinlock spins until the spinlock is free
- Obviously, spinlocks are only useful on SMP
- For synchronising with ISR, there are “interrupt disabling” versions of the spinlock primitives
  - `spin_lock(lock)`, `spin_unlock(lock)`
  - `spin_lock_irq(lock)`, `spin_unlock_irq(lock)`
  - `spin_lock_irqsave(lock, flags)`, `spin_unlock_irqrestore(lock, flags)`

Latency in Multithreaded Kernels

- Non-preemptable sections latency is similar to traditional monolithic kernels
  - L^{np} is bounded by the maximum time spent in KS
- A multithreaded kernel can be made *preemptable* (spinlocks ensure proper synchronisation)
  - `spin_lock()` increases a *preemption counter*
  - `spin_unlock()` decreases a preemption counter; when such counter is 0 the scheduler is invoked to check if a preemption is needed
  - $\Rightarrow$ Can return to US earlier to decrease the latency
- In a preemptable kernel, L^{np} is upper bounded by the maximum size of a kernel critical section
  - Similar to real-time executives

Latency in a Preemptable Kernel



μKernels

- Basic idea: simplify the kernel
  - Reduce to the minimum the number of abstractions exported by the kernel
    - Address Spaces
    - Threads
    - IPC mechanisms (channels, ports, etc...)
  - Most of the “traditional” kernel functionalities implemented in user space
  - Even device drivers can be in user space
- Interactions via IPC (IRQs to drivers as messages, ...)
- Servers: US processes implementing OS functionalities
  - Single-server OSs
  - Multi-server OSs

μKernels: a Failed Experiment?

- First generation of μKernels: Mach, Chorus, ...:
  - Reduced functionalities, but not small (example: Mach is quite big!)
  - Bad performance (need for in-kernel drivers, colocated servers, etc...)
- None of the major OSs is based on a μKernel structure
  - Windows NT used to be based on a μKernel, but now uses drivers running in the kernel address space (colocated servers)
  - MacOS X is based on Mach, but includes FreeBSD functionalities in kernel code
  - Linux is a multithreaded monolithic kernel

μKernels vs Multithreaded Kernels

- μKernels are known to be “more modular” (servers can be stopped / started at run time)
- All the modern monolithic kernels provide a *module* mechanism
- Modules are linked into the kernel, servers are separate programs running in US
- Key difference between μKernels and traditional kernels: each server runs in its own address space
- In some “μKernel systems”, some servers share the same address space for some servers to avoid the IPC overhead
- What’s the difference with multithreaded monolithic kernels?

Latency in μKernel-Based Systems

- Non-preemptable sections latency is similar to monolithic kernels
  - L^{np} is upper-bounded by the maximum amount of time spent in the μKernel
  - μKernels are simpler than monolithic kernels
  - System calls and ISRs should be shorter $\Rightarrow$ the latency in a μKernel should be smaller than in a monolithic kernel
- Unfortunately, the latency reduction achieved by the μKernel structure is not sufficient for real-time systems
  - μKernels have to be modified like monolithic kernels for obtaining good real-time performance

2^{nd} Generation μKernels

- Problems with Mach-like “fat μKernels”
  - The kernel is too big $\rightarrow$ does not fit in cache memory
  - Unefficient IPC mechanisms
- Second generation of μKernels (“MicroKernels Can and Must be Small”): L4
  - Very simple kernel (only few syscalls)
  - Small (fits in cache memory)
  - Super-optimized IPC (not designed to be powerful, but to be efficient)
- The Linux kernel has been ported to L4 (l4linux), and only shows 10% performance penalty
- Real-time performance: bad. The kernel has to be heavily modifies to provide low latencies (Fiasco)

L4Linux and Real-Time

- Idea: a μKernel is so simple and small that it does not need to be preemptable
  - False: Fiasco needed some special care to obtain good real-time performance
- l4linux: single-server OS, providing the Linux ABI
  - Linux applications run unmodified on it
  - Actually the server is the Linux kernel (ported to a new “l4” architecture)
- Real-Time OS: DROPS
  - Non real-time applications run on l4linux
  - Real-time applications directly run on L4
  - The l4linux server should not disable interrupts, or contain non-preemptable sections

"Tamed" L4Linux

- The Linux kernel often disables interrupts (example: `spin_lock_irq()`) or preemption...
- ...So, l4linux risks to increase the latency for L4...
- Solution: in the "L4 architecture", interrupt disabling can be remapped to a *soft interrupt disabling*
  - l4linux disables interrupts $\rightarrow$ no real `cli`
  - IPCs notifying interrupts to l4linux are disabled
  - When l4linux re-enables interrupts, pending interrupts can be notified to the l4linux server via IPC
- As a result, L^{np} is high for the l4linux server (and for Linux applications), but is very low for L4 applications
  - l4linux cannot affect the latency experienced by L4 applications

Dual Kernel Approach

- Idea: Linux applications are non real-time; real-time applications run at lower level
- Try to mix the real-time executive approach with the monolithic approach
  - A Low-level real-time kernel runs at low level and directly handle interrupts and manage the hardware
  - Non real-time interrupts are forwarded to the linux kernel only when they do not interfere with real-time activities
  - Linux cannot disable interrupts (no `cli`), but can only disable (or delay) the forwarding of interrupts from the low-level real-time kernel
- Real-time applications cannot use the Linux kernel

RTLinux, RTAI & Friends - I

- Dual kernel approach: initially used by RTLinux
  - Patch for the Linux kernel to intercept the interrupts
  - Small module implementing a real-time executive
    - Intercept interrupts; handle real-time interrupts with low latency
    - Forward non real-time interrupts to Linux
    - Provide real-time functionalities (POSIX API)
  - Real-time applications are kernel modules
- There is a patent on interrupt forwarding ???
  - RTAI: “Free” implementation of a dual-kernel approach
  - Better maintained than RTLinux
  - Real-time applications are Linux modules: must have an (L)GPL compatible license

RTLinux, RTAI & Friends - II

- I-Pipes: Interrupt Pipelines
  - A small *nanokernel* handles interrupts by sending them to pipelines of applications / kernels that actually manage them
  - Real-time application come first in the pipeline
  - Same functionalities as RTLinux interrupt forwarding
- Described in a paper that has been published before the RTLinux patent $\rightarrow$ patent free
- Adeos nanokernel: implements the interrupt pipelines, similarly to the RTLinux patch
- Xenomai: similar to RTAI; based on Adeos
  - Provides different real-time APIs
  - Allows some form of real-time in US

Other Real-Time Extensions to Linux

- Real-Time performance to Linux processes $\Rightarrow$ need to reduce L^{np} for the Linux kernel, not for low-level applications running under it
- Linux is a multithreaded kernel $\Rightarrow$ need:
  1. Fine-grained locking
  2. Preemptable kernel
  3. Schedulable ISRs and BHs $\Rightarrow$ threaded interrupt handling
  4. Replacing spinlocks with mutexes
  5. A real-time synchronisation protocol to avoid priority inversion
- Remember that current Linux kernels (2.6.21) already provide high-resolution timers

Using Threads for BHs and ISRs

- Using threads for serving BHs and ISRs, it is possible to schedule them
- The priority of interrupts not needed by real-time applications can be decreased, to reduce L^{np}
- Non-threaded ISRs $\Rightarrow$ spinlocks must be used for protecting internal data structures accessed by the ISR
  - The ISR executes in the interrupted process context $\Rightarrow$ it cannot block
- When using threaded ISRs, a lot of spinlocks can be replaced by mutexes
- Spinlocks implicitly use NPP, mutexes do not use any real-time synchronisation protocol
  - At least PI is needed

Ingo Molnar's Realtime-Preempt Tree

- The features presented in the previous slides can surprisingly be implemented with a fairly small kernel patch
- Ingo Molnar maintains the realtime-preempt patch, which is about 1.2MB of code
- Most of the code is needed for changing spinlocks in mutexes
- Various real-time features can be enabled / disabled at kernel configuration time
- The **worst case** total kernel latency is less than $50\mu s$
  - Remember: it was more than $10ms$ on a stock kernel