Real-Time OS Kernels

Luca Abeni
luca.abeni@unitn.it

December 15, 2014

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

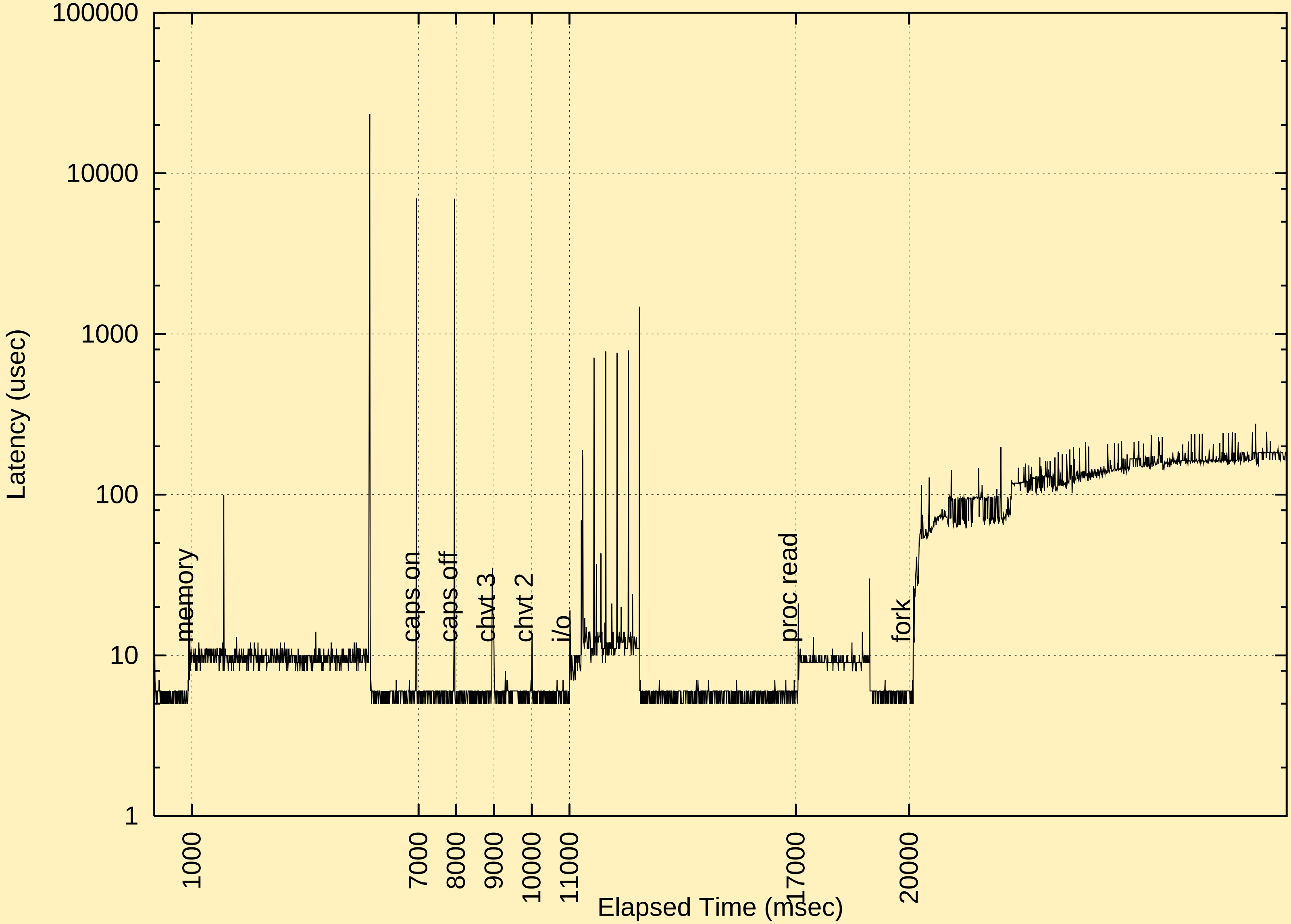
Evolution of the Monolithic Structure

- 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

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 occurr
  - Check for preemptions when exiting kernel's critical sections
- In a preemptable kernel, L^{np} is upper bounded by the maximum size of a kernel critical section
- NOTE: critical section $=$ non-preemptable code... This is NPP!!!

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 vs Multi-server OSs

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...
  - ...But μKernels are simpler than monolithic kernels!
  - System calls and ISRs should be shorter $\Rightarrow$ the latency in a μKernel is generally smaller than in a monolithic kernel
- Unfortunately, the latency reduction achieved by the μKernel structure is often not sufficient for real-time systems
  - Even 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 (designed to be efficient, not powerful)
- Linux ported to L4 (l4linux): only 10% performance penalty
- Real-time performance: not so good. L4 heavily modified (introducing preemption points) to provide low latencies (Fiasco)

L4Linux and Real-Time

- 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 (regular Linux applications)
  - Real-time applications directly run on L4
  - The l4linux server should not disable interrupts, or contain non-preemptable sections
- Use HLP instead of NPP

"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 and real-time ones (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 → patent free
- Adeos nanokernel: implements interrupt pipelines (similar to RTLinux)
- Xenomai: similar to RTAI; based on Adeos
  - Provides different real-time APIs
  - Allows some form of real-time in US

Summing Up...

- Monolithic kernel: high latencies (no real-time)
- Preemptable kernel: kernel critical sections $\rightarrow$ Use NPP to protect them
  - Upper bound for L^{np}, but might be too high (remember the NPP issue)
- μkernel based systems and dual-kernel systems: use HLP instead of NPP
  - HLP requires to know in advance which tasks will use a resource
  - Distinction between real-time and non real-time tasks!
- Can we do better? Priority Inheritance???

Real-Time in Linux User Space

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

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

The Preempt-RT Patch

- The features presented in the previous slides can surprisingly be implemented with a fairly small kernel patch
- Preempt-RT patch, started by Ingo Molnar and other Linux developers; now maintained by Thomas Gleixner
- `https://www.kernel.org/pub/linux/kernel/projects/rt`: about 700KB 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