Real-Time OS Kernels
Real Time Operating Systems and Middleware

Luca Abeni
luca.abeni@unitn.it
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
Real-Time Executives - 2

● Advantages:
  ● Simple, small, low overhead
  ● Only the needed code is linked in the final image

● Disadvantages:
  ● No protection
  ● Applications can even disable interrupts → $L^{np}$
    risks to be unpredictable
Real-Time Executives - 3

- Consistency of the internal structures is generally ensured by disabling interrupts
  - $L^{np}$ is bounded by the maximum amount of time interrupts are disabled
  - ...Disabled by the executive or by applications!!!
- Generally used only when memory footprint is important, or when the CPU does not provide a privileged mode
  - Example: TinyOS [http://www.tinyos.net](http://www.tinyos.net)
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
Interrupt handlers split in two parts

- Short and fast ISR
- "Soft IRQ handler"

Soft IRQ handler: *deferred* handler

- Traditionally known ass 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 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 → 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???
• 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: μkernels or dual-kernel systems
Basic idea: simplify the kernel
- Reduce to 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
  • OS kernel as a single user-space process: Single-server OSs
  • Multiple user-space processes (a server per driver, FS server, network server, ...): 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
Latency in $\mu$ Kernel-Based Systems - 1

- Non-preemptable sections latency is similar to monolithic kernels
  - $L^{np}$ is upper-bounded by the maximum amount of time spent in the $\mu$ Kernel...
  - ...But $\mu$ Kernels are simpler than monolithic kernels!
  - System calls and ISRs should be shorter $\Rightarrow$ the latency in a $\mu$ Kernel is generally smaller than in a monolithic kernel
Latency in $\mu$Kernel-Based Systems - 2

- Unfortunately, the latency reduction achieved by the $\mu$Kernel structure is often not sufficient for real-time systems.
- Even $\mu$Kernels have to be modified like monolithic kernels for obtaining good real-time performance.
- $(\mu)$kernel preemptability, ...
2nd Generation \( \mu \)Kernels

- Problems with Mach-like “fat \( \mu \)Kernels”
  - The kernel is too big \( \rightarrow \) does not fit in cache memory
  - Unefficient IPC mechanisms
- Second generation of \( \mu \)Kernels (“MicroKernels Can and Must be Small”): \textbf{L4}
  - Very simple kernel (only few syscalls)
  - Small (fits in cache memory)
  - Super-optimized IPC (designed to be efficient, not powerful)
2nd Generation $\mu$Kernels: Performance

- L4 $\mu$kernel: optimised for performance
  - Impact on global OS performance?
  - Real-Time performance?
- Linux ported to L4: l4linux
  - Single-Server OS
  - Only 10% performance penalty!
- Real-time performance: not so good. L4 heavily modified (introducing preemption points) to provide low latencies (Fiasco)
L4Linux

- 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)
- Idea: a $\mu$Kernel 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 and Real-Time

- 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
  - Easy to identify RT tasks: native L4 tasks!
  - The l4linux server must never have a priority higher than RT applications
“Tamed” L4Linux - 1

- 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 → 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
l4linux does not really disable hw interrupts

- $L^{np}$ 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

- HLP requires to know which applications use the resource...
- In this context, it means “which applications use l4linux”
Dual Kernel Approach

- HLP idea: Linux applications are non real-time; real-time applications run at lower level
- Instead of using $\mu$ kernels, mix the real-time executive approach with the monolithic approach
  - Low-level real-time kernel: directly handles interrupts and manages the hardware
  - Non real-time interrupts: forwarded to Linux only when they do not interfere with RT activities
  - Linux cannot disable interrupts (no `cli`)
  - Can only disable (or delay) interrupt forwarding
- Real-time applications cannot use the Linux kernel
RTLinux

- Dual kernel approach: initially used by RTLinux
  - Patch for the Linux kernel to intercept the interrupts
  - Small kernel module implementing a real-time executive
    - Handle real-time interrupts (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 ????
RTLinux & RTAI

- 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
- No problem in Europe, maybe subject to RTLinux patent in the US
  - Big problem for adoption in the industry
  - Would you use something that might be infringing a patent?
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, but different naming!

- Described in a paper that has been published before the RTLinux patent → patent free
I-Pipes Implementation

- Adeos nanokernel: implements interrupt pipelines
  - Same functionalities as RTLinux, but patent-free!
  - Can be optionally used by RTAI
- Xenomai: similar to RTAI; based on Adeos
  - Provides different real-time APIs
- Xenomai 3: both dual-kernel and user-space
Summing Up...

- Monolithic kernel: high latencies (no real-time)
- Preemptable kernel: kernel critical sections → Use NPP to protect them
  - Upper bound for $L_{np}$, but might be too high
- μkernel and dual-kernel: use HLP instead of NPP
  - HLP requires to know in advance which tasks will use a resource
  - Distinction between RT and NRT tasks!
- Can we do better? How to use PI???
HLP Idea: do not care about Linux kernel latencies, but make sure that they do not affect RT tasks

- RT tasks: not Linux tasks!

Real-Time performance to Linux processes $\Rightarrow$ need to reduce $L^{np}$ for the Linux kernel, not for low-level applications running under it

- How to reduce $L^{np}$? Using PI directly is not easy...

  - There is a reason for using NPP
  - In some situations, the kernel cannot block!
  - But PI is a blocking protocol...
RT in User Space: Requirements

- 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 (PI) for these mutexes

- 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 handlers: ISRs and BHs always preempt all tasks!!!
    - NRT tasks can trigger high latencies by just doing a lot of I/O!!!
  - Threaded handlers: if an interrupt is not needed by RT tasks, its priority can be lower than all the RT tasks priorities
Threaded Interrupt Handlers and PI

- Non-threaded ISRs ⇒ use spinlocks to protect data structures accessed by the ISR
  - The ISR executes in the interrupted process context ⇒ it cannot block
- Using threaded ISRs, spinlocks can be replaced with mutexes
- Spinlocks implicitly use NPP, mutexes can use PI!!!
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
- Most of the code just changes spinlocks in mutexes
- Various real-time features can be enabled / disabled at kernel configuration time
Preempt-RT: Performance

- Continuous Integration and testing:
  https://www.osadl.org/QA-Farm-Realtime.qa-farm-about.0.html

- On a standard PC, **Worst Case** kernel latency less than 50\(\mu\)s
  - Remember: it was more than 10\(ms\) on a vanilla kernel!

- Much more tested than many other “RT” kernels
  - Long (continuous!) runs
  - Multiple CPUs / architectures