Developing Real-Time Applications

Real Time Operating Systems and Middleware

Luca Abeni

`luca.abeni@unitn.it`

Real-Time Applications

- Characterised by temporal constraints
  - deadlines
- Concurrent (application: set of real-time tasks)
  - Threads
  - Processes
  - Cyclic Executinve...
- Periodic, sporadic, or aperiodic behaviour

Requirements

- Need to implement periodic behaviour
  - Requirements on the API
- Need an appropriate scheduling policy
  - Again, requirements on the API...
  - Also requirements on kernel latency!
- Latency requirements:
  - Requirements on kernel structure
  - Need to disable “lazy” behaviours (example: lazy memory allocation)

Programming Interface

- Real-time applications can use a standard API
  - POSIX
- ...Or some specialised (worse: proprietary) one!
  - RTAI
  - vxworks, etc...
- Xenomai provides *skins* providing support for existing real-time APIs
  - Easy porting from non-standard APIs

Kernel Structure

- Difference between real-time kernels and non real-time ones:
  - Real-time kernels provide deterministic (and low!) latency
- Some possible technologies
  - Distinction between real-time and non real-time tasks (HLP)
  - No distinction between real-time and non real-time tasks

Using HLP, NPP or PI

- Real-Time applications do not share the kernel with non real-time ones:
  - μkernels
  - Dual-kernel systems
- Real-Time applications share the kernel with non real-time ones:
  - Preemptable kernels (NPP)
  - Preempt-RT (PI)

μkernels and Dual-Kernel Systems - 1

- Basic Idea: real-time applications do not use the Linux (or similar) kernel
  - Kernel critical sections cannot cause latencies on real-time applications
  - So, Linux can have large critical sections
- Real-Time applications use a “lower level kernel”
  - Can be a μkernel or some kind of real-time executive living in kernel space
  - Real-Time applications run in user space (μkernel) or in kernel space (dual kernel)

μkernels and Dual-Kernel Systems - 2

- Real-Time applications have higher priorities than non real-time ones
  - Even higher priority than the kernel!
- The “lower level kernel” has short critical sections
  - “Lower level kernel” critical sections: non preemptable
  - Linux critical sections: non preemptable by Linux (and by non real-time applications) but preemptable by real-time applications
    - HLP!!!

Running all the Applications on the same Kernel

- No explicit distinction between real-time and non real-time applications
  - All the applications use the same kernel (Linux)
  - Impossible to use HLP
- Need to reduce the size of critical sections...
- ...And to use an appropriate resource sharing protocol
  - Spinlocks can only use NPP
  - To use PI, we need mutexes $\Rightarrow$ IRQ threads, etc...

Real-Time in User Space

- Address space protection
- Applications cannot disable interrupts
- Real-time applications can use the kernel (or μkernel) functionalities
  - System libraries (or `libc`) to invoke syscalls
  - μkernel: maybe some additional user-space servers
- Running on Linux: can use drivers, network, etc...
- μkernels: must re-implement everything

Real-Time in Linux User Space

- Real-time over linux: preemptable kernel (NPP) or Preempt-RT (PI)
  - Can use Linux functionalities (drivers, network stack, filesystem, ...)
  - Cost: to get low latencies, Linux has to be modified
- NPP: long critical sections in the kernel can cause high latencies on all the tasks
- PI: tasks not using a critical section are not penalised

Linux Real-Time Applications

- PI / Preempt-RT: Properly written real-time tasks experience good performance
- Some special care is needed
  - Low latency: lower-priority tasks cannot affect the response time....
  - ...But real-time tasks still need “some tricks” to achieve deterministic response times
- Example: virtual memory management / dynamic memory allocation...

Virtual Memory Management

- User applications do not access physical memory
- Virtual memory address space: divided in *pages*
  - A page in virtual memory can be mappedo to a page in physical memory...
  - ...Or can have no correspondent page in physical memory
  - In the first case, the memory can be accessed
  - In the second case, it must be mapped in physical memory first
- When does the mapping happen?

Page Faults

- When a process starts, no page is mapped in physical memory
- When a non-mapped page is accessed, page fault
  - A kernel handler is invoked
  - Finds a free physical page...
  - ...And creates the mapping!
- What happens if no free physical page is found?
  - Some existing mapping is removed
  - To avoid data loss, the content of the page is saved to disk

Minor and Major Faults

- A page fault can happen because:
  - This is the first time we access a memory page (minor fault)
  - We access a page that has been previously unmapped / swapped to disk (major fault)
- Majour faults are (obviously) more expensive
- All page faults make the memory access time less predictable
- Should be avoided in real-time applications

Avoiding Major Faults

- How to avoid Majour Faults? Just don’t swap pages to disk!
  - If a mapped page cannot be unmapped...
  - ...Then no major fault can happen on it!
- POSIX provides system calls (`mlockall()`, ...) to “pin” memory pages in physical memory
- A “pinned” page cannot be unmapped / swapped to disk
- Only one minor fault the first time we access the page

What About Minor Faults?

- Minor faults cannot be avoided
  - To access a memory page, it must be mapped...
- The problem is the “lazy” memory management used by many OS
  - A memory page is mapped when it is accessed the first time
  - Minor fault on the first access
  - Add unpredictability to memory access time!
- Solution: map all the pages before starting real-time activities

Minor Faults in Real-Time Applications

- We do not want page faults during real-time activities
  - Majour faults: use `mlockall()`
  - Minor faults: force mapping all the pages in an *initialisation phase*
- All page faults in initialisation $\rightarrow$ no minor faults during real-time activities
- Touch all the memory buffers before starting
- Example: allocate dynamic memory before the first job, and memset it to 0 immediately

Dynamic Memory Allocation, Again...

- `malloc()` & friends are safe only during initialisation...
- What to do if the jobs of a real-time taks need to dynamically allocate / free memory?
- Dirty trick: play with the memory allocation subsystem so that freed memory is not returned to the kernel
  - Example: use `mallopt()` or similar
  - A sufficient amount of memory needs to be allocated and freed during initialisation

Real-Time in Kernel Space

- Dual-kernel approach (RTLinux, RTAI, Xenomai, ...)
  - Real-Time applications do not use the Linux kernel
    - So, we can use HLP!
  - Real-Time task: kernel thread
    - So, real-time applications run in kernel space!!!
- On Linux: use kernel modules

Linux Kernel Modules

- Kernel module: code that can be dynamically loaded/unloaded into the kernel at runtime
- Change the kernel code without needing to reboot the system
- More technically: the modules' object code is dynamically linked to the running kernel code
  - Form of dynamic linking!
- This mechanism can be used to load kernel-space real-time applications!

Using Kernel Modules

- Kernel Module: *kernel object* $\rightarrow$ `.ko` file
- Inserted with `modprobe <module name>`
- Can be removed with `rmmod <module name>`
- When inserted, a kernel module can:
  - Register some services
  - Start some tasks (kernel threads)
- A kernel module can use some *exported kernel functions*

Kernel Programming - 1

- No single entry point (no “`main()` function)
- No memory protection
  - Kernel Memory Address Space: all the memory can be accessed
  - Kernel-space tasks can easily corrupt important data structures!
- Not linked to standard libraries
  - Cannot include `<stdio.h>` and friends...
  - No standard C library!

Kernel Programming - 2

- The kernel (or nanokernel, or ...) provides some functions we can use
  - Example, no `printf()`, but `printk()`...
- Errors do not result in segmentation faults...
- ...But can cause system crashes!
- Other weird details
  - No floating point (do not use `float` or `double`)
  - Small stack ($4KB$ or $8KB$)
  - Atomic contexts, ...

Kernel Programming Language

- OS kernels are generally coded in C or C++
  - The Linux kernel uses C
  - Subset of C99 + some extensions (`likely()` / `unlikely()` annotations, etc...)
- As said, no access to standard libraries
  - Different set of header files and utility functions
- Some Assembly is used (for entry points, etc...)
- Example: Linked Lists (`include/linux/list.h`)

Writing Linux Kernel Modules

- Written in C99 + extensions (see previous slide)
- Must include some headers:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
```

- Must define two entry points: *init* and *cleanup*
  - Init entry point: called when the module is inserted
  - Cleanup entry point: called when the module is removed

The Init Entry Point

```
static int __init my_init(void)
{
  ...
  return 0;
}

module_init(my_init);
```

- `static`: not used outside this compilation unit
- `__init`: annotation for the kernel (not used after `insmod`)
- `return 0;`: module initialised without errors
- `module_init(my_init);`: mark `my_init` as the init entry point

The Exit Entry Point

```
static void __exit my_cleanup(void)
{
    ...
}

module_exit(my_cleanup);
```

- `__exit`: annotation for the kernel (used only in `rmmod`)
- `module_exit(my_cleanup);`: mark `my_cleanup` as the cleanup entry point
- Responsible for undoing things done by init
- If not defined, the module cannot be unloaded

Applications as Kernel Modules

- The init entry point must return quickly
  - `modprobe` does not terminate until init returns
- Just creates some (real-time!) threads and return
  - After loading the module, the application is started!
- The cleanup entry point stops the threads
- See Xenomai example (in the lab!)