Real-Time and Embedded Guide

Herman Bruyninckx
K.U.Leuven, Mechanical Engineering

Leuven
Belgium
Herman.Bruyninckx@mech.kuleuven.ac.be

Real-Time and Embedded Guide

by Herman Bruyninckx

This Guide covers the fundamentals of (i) real-time and embedded operating systems (focusing mostly on the differences with general purpose operating systems such as Linux), and (ii) real-time programming. The emphasis is on Free Software and Open Source Software examples: RTAI, RTLinux, eCos, RT-EMS, uCLinux, . . . , with a more than proportional focus on RTAI.

This text also talks about design issues, software patterns and frameworks for real-time applications. That is, the "high-level" aspects of these software projects. These higher levels are often poorly dealt with in publications on real-time programming, which leads to the unfortunate situation that still too many real-time programmers use *only* the powerful but dangerously unstructured API of their RTOS. Missing the chance to develop more structured, and, hence, more deterministic and more portable software systems.

Both the low-level RTOS primitives, and the high-level design issues, are illustrated by the real-world example of a hard real-time core for feedback control and signal processing.

Revision History

Revision 0.01	Aug 31, 2000	Revised by: hb
Initial draft		
Revision 0.02	Sep 30, 2000	Revised by: hb
Added: more info about signals		
Revision 0.03	Sep 20, 2001	Revised by: hb
Removed: empty hardware, user space GUI and FAQ sections. Added: Software Patterns		
Revision 0.04-build-20021211-1936	Dec., 11 2002	Revised by: hb
Extended and heavily reworked version. Preparing for pre-release.		

Table of Contents

List of Figures

About this Guide

1. Purpose and scope

This Guide consist of several parts: Part 1 provides a *top-down overview* of real-time and embedded operating systems, up to a more detailed description of the features and implementation of a “typical” RTOS, i.e., RTAI; Part 2 gives more details about implementation of real-time functionality. Part 3 introduces some time-proven design solutions to common problems in real-time programming, as well as a list of design and programming hints, to help readers gain time and reliability in their designs and implementations.

The top-down view on real-time and embedded operating systems is complementary to the typical *bottom-up* “show-me-the-code” and “bazaar” approach of development and documentation writing in the free software world. Not that there is something wrong with this approach, but this Guide’s goal is different: it wants to make it easier for newcomers to grasp the basic concepts and techniques behind real-time operating systems, and to help them see the forest for the trees, without having to go and read the code in a ton of different files. Nevertheless: the source code of the presented projects remains the *only* complete and up-to-date documentation.

The document tries to be as independent as possible of any particular implementation or project: so, the concepts introduced in the theoretical part of the text are not necessarily available in each and every concrete example given in the text. Moreover, this Guide is not meant to be an exhaustive textbook on real-time programming, so the reader is urged to go and read some operating system textbooks, and other “low-level” books such as the Linux Device Drivers (http://www.xml.com/ldd/chapter/book/index.html) book, for missing details. The reader should be familiar with the basics of operating systems, and be able to read C code.

This guide first explains the general principles behind real-time and embedded operating systems. These principles are not too different from general operating systems such as Linux or Microsoft NT, but the emphasis lies on *maximum determinism*, and not on *maximum average throughput*. Because determinism is often compromised in “high-level” programming language and operating system constructs, real-time designers are confronted more directly than “normal” application developers with concepts, timing, memory, and efficiency at the level of the operating system.

Another primary goal of this Guide is educational: it could over time evolve into classroom notes, featuring demos, annotated code, more graphical illustrations, and more examples of good design and code. Whether it will reach these educational goals depends on *your* contributions, as critical reader of the text, looking out for opportunities to help improve the free documentation available on your favourite free software project. . .

2. Feedback

This Guide still has a number of paragraphs marked with *"TODO"*, signalling parts that are still to be filled in with more details, or where the current treatment needs more thought and/or information.

Please direct all questions, additions or suggestions for changes to <`Herman.Bruyninckx@mech.kuleuven.ac.be`>.

3. Copyrights, Licenses and Trademarks

4. Acknowledgements

Large parts of this document were written with financial support from the Flemish *Fonds voor Wetenschappelijk Onderzoek (FWO)*, and the *Katholieke Universiteit Leuven*, Belgium. The hospitality offered by Prof. Henrik Christensen of *Kungliga Tekniska Högskolan (KTH)* in Stockholm, where other parts of this document were written, is gratefully acknowledged.

The style for this Guide was originally copied from the LDP Author Guide (http://www.linuxdoc.org/LDP/LDP-Author-Guide/) written by Mark F. Komarinski and Jorge Godoy. It used the ldp.dsl SGML stylesheet from the Linux Documentation Project (http://www.linuxdoc.org). The current style is *DocBook*.

Linux and the indispensable GNU libraries and tools are wonderful gifts from Linus Torvalds, the people at the Free Software Foundation (http://www.fsf.org), and thousands of others. While, in general, *GNU/Linux* is the appropriate name to denote the popular free software operating system, this text usually talks about Linux, because the *kernel* is the topic of this document. The text also uses the term *free software* as a general name for software released under licences approved by both the Free Software Foundation (http://www.fsf.org/philosophy/license-list.html) and the Open Source Initiative (http://www.opensource.org/licenses/).

The RTLinux (http://www.rtlinux.com) real-time extensions to Linux were originally created by Victor Yodaiken and Michael Barabanov. Paolo Mantegazza created the Real Time Application Interface (http://www.rtai.org) (RTAI) real-time extensions. Karim Yaghmour come up with the design of an alternative approach towards building a real-time nano-kernel underneath Linux (or any operating system kernel for that matter); this design was implemented in the Adeos project by Philippe Gerum. This text's discussion on real-time device drivers is much inspired by David Schleef's design for Comedi (http://stm.lbl.gov/comedi/).

Klaas Gadeyne and Patrice Kadionik gave valuable feedback on the first draft. Error feedback was also received from Werner Spielmann, Gregory Matus, and Oscar Esteban. Paolo Mantegazza and Philippe Gerum helped to clarify some RTAI internals. Many thanks also go to Peter Soetens of the Orocos project (http://www.orocos.org), and the many critical students and summer interns whose questions stimulated me to look deeper into all those things that I thought I understood but didn't.

I. Operating system basics

This Part introduces the concepts and primitives with which general purpose as well as real-time and embedded operating systems are built. The text dicusses the applicability and appropriateness of all these concepts in real-time and embedded operating systems.

(TODO: more annotated code examples.)

Chapter 1. Real-time and embedded operating systems

This Chapter discusses the basics of operating systems in general, and real-time and embedded operating systems in particular. (This text uses the abbreviations OS, RTOS and EOS, respectively.) This discussion makes clear why standard Linux doesn't qualify as a real-time OS, nor as an embedded OS.

Real-time and embedded operating systems are in most respects similar to general purpose operating systems: they provide the interface between application programs and the system hardware, and they rely on basically the same set of programming primitives and concepts. But general purpose operating systems make different trade-offs in *applying* these primitives, because they have different goals.

1.1. OS responsibilities

This Section discusses the basic responsibilities of the operating system that are relevant for this text: (i) task management and scheduling, (ii) (deferred) interrupt servicing, (iii) inter-process communication and synchronization, and (iv) memory management. General-purpose operating systems also have other responsibilities, which are beyond the horizon of a *real-time* operating system: file systems and file management, (graphical) user interaction, communication protocol stacks, disk IO, to name a few. More details about the relevant responsibilities are given in the following Chapters.

1.1.1. Task management and scheduling

Task (or "process", or "thread") management is a primary job of the operating system: tasks must be created and deleted while the system is running; tasks can change their priority levels, their timing constraints, their memory needs; etcetera. Task management for an RTOS is a bit more dangerous than for a general purpose OS: if a real-time task is created, it *has* to get the memory it needs without delay, and that memory *has* to be locked in main memory in order to avoid access latencies due to swapping; changing run-time priorities influences the run-time behaviour of the whole system and hence also the predictability which is so important for an RTOS. So, dynamic process management is a potential headache for an RTOS. Chapter 2 gives more details.

In general, multiple tasks will be active at the same time, and the OS is responsible for sharing the available resources (CPU time, memory, etc.) over the tasks. The CPU is one of the important resources, and deciding how to share the CPU over the tasks is called "scheduling".

The general trade-off made in scheduling algorithms is between, on the one hand, the *simplicity* (and hence efficiency) of the algorithm, and, on the other hand, its *optimality*. (Note that various optimality criterions exist!) Algorithms that want to be globally optimal are usually quite complex, and/or require knowledge about a large number of task parameters, that are often not straightforward to find on line (e.g., the duration of the next run of a specific task; the time instants when sleeping tasks will become ready to run; etc.). Real-time and embedded operating systems favour simple scheduling algorithms, because these take a small and deterministic amount of computing time, and require little memory footprint for their code.

General purpose and real-time operating systems differ considerably in their scheduling algorithms. They use the same basic principles, but apply them differently because they have to satisfy different performance criterions. A general purpose OS aims at maximum *average* throughput, a real-time OS aims at *deterministic* behaviour, and an embedded OS wants to keep memory footprint and power consumption low. A large variety of “real-time” scheduling algorithms exists, but some are standard in most real-time operating systems (see Section 2.5): *static priority scheduling*, *earliest deadline first (EDF)*, and *rate-monotonic scheduling*.

1.1.2. Interrupt servicing

An operating system must not only be able to schedule tasks according to a deterministic algorithm, but it also has to service peripheral hardware, such as timers, motors, sensors, communication devices, disks, etc. All of those can request the attention of the OS *asynchronously*, i.e., at the time that *they* want to use the OS services, the OS has to make sure it is ready to service the requests. Such a request for attention is often signaled by means of an *interrupt*. There are two kinds of interrupts:

- *Hardware interrupt.* The peripheral device can put a bit on a particular hardware channel that triggers the processor(s) on which the OS runs, to signal that the device needs servicing. The result of this trigger is that the processor saves its current state, and jumps to an address in its memory space, that has been connected to the hardware interrupt at initialisation time.
- *Software interrupt.* Many processors have built-in software instructions with which the effect of an hardware interrupt can be generated in software. The result of a software interrupt is also a triggering of the processor, so that it jumps to a pre-specified address.

The operating system is, in principle, not involved in the execution of the code triggered by the hardware interrupt: this is taken care of by the CPU without software interference. The OS, however, does have influence on (i) connecting a memory address to every interrupt line, and (ii) what has to be done *immediately after* the interrupt has been serviced, i.e., how *“deferred interrupt servicing”* is to be handled. Obviously, real-time operating systems have a specific approach towards working with interrupts, because they are a primary means to guarantee that tasks gets serviced deterministically. Chapter 3 gives more details.

1.1.3. Communication and synchronization

A third responsibility of an OS is commonly known under the name of *Inter-Process Communication* (IPC). (“Process” is, in this context, just another name for “task”.) The general name IPC collects a large set of programming primitives that the operating system makes available to tasks that need to exchange information with other tasks, or synchronize their actions. Again, an RTOS has to make sure that this communication and synchronization take place in a deterministic way. Chapter 4 gives more details.

Besides communication and synchronization with other tasks that run on the same computer, some tasks also need to talk to other computers, or to peripheral hardware (such as analog input or output cards). This involves some peripheral hardware, such as a serial line or a network, and special purpose device drivers (Chapter 7).

1.1.4. Memory management

A fourth responsibility of the OS is *memory management*: the different tasks in the system all require part of the available memory, often to be placed on specified hardware addresses (for memory-mapped IO). The job of the OS then is (i) to give each task the memory it needs (*memory allocation*), (ii) to map the real memory onto the address ranges used in the different tasks (*memory mapping*), and (iii) to take the appropriate action when a task uses memory that it has not allocated. (Common causes are: unconnected pointers and array indexing beyond the bounds of the array.) This is the so-called *memory protection* feature of the OS. Of course, what exactly the "appropriate action" should be depends on the application; often it boils down to the simplest solution: killing the task and notifying the user. Chapter 6 gives more details.

1.2. Trade-offs

This Section discusses some of the trade-offs that (both, general purpose, and real-time and embedded) operating system designers commonly make.

- *Kernel space versus user space versus real-time space*.

  Most modern processors allow programs to run in two different hardware protection levels. Linux calls these two levels *kernel space* and *user space*. The latter have more protection against erroneous accesses to physical memory of I/O devices, but access most of the hardware with larger latencies than kernels space tasks. The real-time Linux variants add a third layer, the *real-time space*. This is in fact nothing else but a part of kernel space used, but used in a particular way.

- *Monolithic kernel versus micro-kernel*.

  A monolithic kernel has al OS services (including device drivers, network stacks, file systems, etc.) running within the *privileged mode* of the processor. (This doesn't mean that the whole kernel is one single C file!) A micro-kernel, on the other hand, uses the privileged mode only for really core services (task management and scheduling, interprocess communication, interrupt handling, and memory management), and has most of the device drivers and OS services running as "normal" tasks. The trade-off between both is as follows: a monolithic kernel is easier to make more efficient (because OS services can run completely without switches from privileged to non-privileged mode), but a micro-kernel is more difficult to crash (an error in a device driver that doesn't run in privileged mode is less likely to cause a system halt than an error occurring in privileged mode).

UNIX, Linux and Microsoft NT have monolithic kernels; QNX, FIASCO, VxWorks, and GNU/Hurd have micro-kernels. Linux, as well as some commercial UNIX systems, allow to dynamically or statically change the number of services in the kernel: extra functionality is added by loading a *module*. But the loaded functionality becomes part of the monolithic kernel. A minimal Linux kernel (which includes memory management, task switching and timer services) is some hundreds of kilobytes big; this approaches the footprint for embedded systems. However, more and more embedded systems have footprints of more than a megabyte, because they also require network stacks and various communication functionalities.

- *Pre-emptable kernel or not*.

  Linux was originally a non-pre-emptable kernel: a kernel space task cannot be interrupted by other kernel space tasks, or by user space tasks. The kernel is "locked" as long as one kernel function is executing. This usage of locks (Section 4.6) makes the design of the kernel simpler, but introduces indeterministic latencies which are not tolerable in an RTOS.

  In the 2.5 kernel series, Linux gets a more and more fine-grained kernel locking mechanism, *and* has become to a large extent pre-emptable. (See Section 2.8.) Linux still has one "Big Kernel Lock (BKL)," called `kernel_flag` in the Linux source code, but now independent subsystems (networking, disk IO, etc.) get their own sets of locks.

- *Scalability*.

  Finer-grained locking is good for *scalability*, but usually an overhead for single-CPU systems. *Solaris* is an example of a very fine-grained and scalable operating system, which performs worse on "low-end" PCs. The Linux Scalability Effort (http://sourceforge.net/projects/lse/) project has more information about the ongoing activities in this area, as far as the Linux kernel is concerned.

  Scalability is *much less* of an issue in *real-time* applications, because the goals are so differen: the desire behind scalable systems is to divide a large work load transparantly over a number of available CPUs, while the desire behind real-time systems is have everything controlled in a strictly deterministic way.

- *Memory management versus shared memory*.

  Virtual memory and dynamic allocation and de-allocation of memory pages are amongst the most commonly used memory management services of a general purpose operating system. However, this memory management induces overhead, *and* some simpler processors have no support for this memory management. On these processors (which power an enormous number of embedded systems!), all tasks share the same memory space, such that developers must take care of the proper use of that memory. Also some real-time kernels (such as RTLinux) have all their tasks share the same address space (even if the processor supports memory management), because this allows more efficient code.

- *Dedicated versus general*.

  For many applications, it is worthwhile not to use a commercially or freely available operating system, but write one that is optimised for the task at hand. Examples are the operating systems for mobile phones, or Personal Digital Assistants. Standard operating systems would be too big, and they don't have the specific signal processing support (speech and handwriting recognition) that is typical for these applications. Some applications even don't need an operating system at all. (For example, a simple vending machine.) The trade-offs here are: cost of development and decreased portability, against cost of smaller and cheaper embedded systems.

- *Operating system versus language runtime*.

  Application programs make use of "lower-level" primitives to build their functionality. This functionality can be offered by the operating system (via system calls), or by a programming language (via language primitives and libraries). Languages such as C++, Ada and Java offer lots of functionality this way: memory management, threading, task synchronization, exception handling, etc. This functionality is collected in a so-called *runtime*. The advantages of using a runtime are: its interface is portable over different operating systems, and it offers ready-to-use and/or safe solutions to common problems. The disadvantages are that a runtime is in general "heavy", not deterministic in execution time, and not very configurable. These disadvantages are important in real-time and embedded contexts.

1.3. Time

Not surprisingly, "time" plays an important role in the design and use of a real-time operating system. This Section introduces some relevant terminology and definitions.

1.3.1. Real time

Probably you'll find as many interpretations of the meaning of *real time* as you find publications on this topic. One simple definition is:

> A real-time operating system is able to execute all of its tasks without violating *specified* timing constraints.

Another definition is:

> Times at which tasks will execute can be *predicted deterministically* on the basis of knowledge about the system's hardware and software.

That means, if the hardware *can* do the job, the RTOS software *will* do the job deterministically. (This determinism must be softened a bit, because of the “stochastic” nature of the inevitable scheduling “jitter”, see Section 1.3.2.)

One often makes distinction between “soft real time” and “hard real time”. “Soft” indicates that not meeting the specified timing constraints is not a disaster, while it *is* a disaster for a hard real-time system. For example: playing an audio or video file is soft real time, because few people will notice when a sample comes a fraction of a second too late. Steering a space probe, on the other hand, requires hard real time, because the rocket moves with a velocity of several kilometers per second such that small delays in the steering signals add up to significant disturbances in the orbit which can cause erroneous atmosphere entry situations. Precision mills and high-accuracy radiation or surgical robots are other examples that require hard real-time: moving the mill or the robot one tenth of a millimeter too far due to timing errors can cause the rejection of produced parts, or the death of patients.

Practically speaking, the distinction between soft and hard real time is often (implicitly and mistakenly) related to the time scales involved in the system: in this reasoning, soft real-time tasks must typically be scheduled with (coarser than) *milli-seconds* accuracy, and hard real-time tasks with *micro-seconds* accuracy. But this implicit assumption has many exceptions! For example, a one-dollar 4 bit processor controlling a traffic light can be more hard real time (in the sense of “deterministic”) than a 5000 dollar Athlon-based e-commerce server.

1.3.2. Latency

The *latency* (or *tardiness*) of a task is the difference between the instant of time on which the task should have started (or finished) and the instant of time on which it actually did. (Or, in different contexts, the time between the *generation* of an event, and its *perception*.) Latencies are due to several factors: (i) the timing properties of processor, bus, memory (on-chip cache, off-chip RAM and ROM) and peripheral devices, (ii) the scheduling properties of the OS, (iii) the *pre-emptiveness* of its kernel, (iv) the load on the system (i.e., the number of tasks that want to be scheduled concurrently), and (v) the *context switch* time. This latter is the time the processor needs to save the data of the currently running task (e.g., registers, stack, and instruction pointer), and to replace it with the local data of the newly scheduled task. Few of these factors are constant over time, and the statistical distribution of the latencies in the subsequent schedulings of tasks is called the *jitter*.

This is a far from exhaustive list of kernel activities that introduce *indeterminism* into the timing behaviour of a (general purpose) operating system:

- *Accessing the hard disk.* Because the alignment of sectors, and the distance between the tracks needed by a given task are variable, that task cannot be sure about how long it will take to access the data it needs. In addition, hard disks are mechanical devices, whose time scales are much longer than purely electronic devices (such as RAM memory); and accesses to the hard disk are *buffered* in order to reduce *average* disk access time.
- *Accessing a network.* Especially with the TCP/IP protocol, that re-sends packets in case of transmission errors.

- *Low-resolution timing.* See Section 1.3.4.
- Another delay related to time keeping is the fact that *programming the timer chip* often generates unpredictable delays. This delay is of the order of microseconds, so only important for really high-accuracy timing.
- *Non-real-time device drivers.* Device drivers are often sloppy about their time budget: they use busy waiting or roughly estimated sleeping periods, instead of timer interrupts, or lock resources longer than strictly necessary, or run in user space with the corresponding timing unpredictability.
- *Memory allocation and management.* After a task has asked for more memory (e.g., through a `malloc` function call), the time that the memory allocation task needs to fulfill the request is unpredictable. Especially when the allocated memory has become strongly fragmented and no contiguous block of memory can be allocated. Moreover, a general purpose operating system swaps code and data out of the physical memory when the total memory requirements of all tasks is larger than the available physical memory.
- *`proc` file system.* This is the very rich (non-graphical) user interface to what is going on inside the Linux kernel: all this information is offered to user tasks in the form of “files” in this (virtual!) file system. However, accessing information in this file system implies significant overhead in some cases, because the files are virtual: they are “created” only when needed.

The exact *magnitude* of all the above-mentioned time delays changes very strongly between different hardware. Hence, it is not just the operating system software that makes the difference. For some applications, the context switch time is most important (e.g., for sampling audio signals at 44kHz), while other applications require high computational performance, at lower scheduling frequencies (e.g., robot motion control at 1kHz). But again, some tasks, such as speech processing, require both.

1.3.3. Timing constraints

Different applications have different timing constraints, which, ideally, the RTOS should be able to satisfy. However, there still doesn’t exist general and guaranteed scheduler algorithms (Chapter 2) that are able to satisfy all the following classes of time constraints:

- *Deadline*: a task has to be completed before a given instant in time, but when exactly the task is performed during the time interval between now and the deadline is not important for the quality of the final result. For example: the processor must fill the buffer of a sound card before that buffer empties; the voltage on an output port must reach a given level before another peripheral device comes and reads that value.
- *Zero execution time*: the task must be performed in a time period that is zero in the ideal case. For example: digital control theory assumes that taking a measurement, caculating the control action, and sending it out to a peripheral device all take place instantaneously.
- *Quality of Service* (QoS): the task must get a fixed amount of “service” per time unit. (“Service” often means “CPU time”, but could also be “memory pages”, “network bandwidth” or “disk access bandwidth”.) This is important for applications such as multimedia (in order to read or write streaming

audio or video data to the multimedia devices), or network servers (both in order to guarantee a minimum service as in order to avoid "denial of service" attacks).

The QoS is often specified by means of a small number of parameters: "s" seconds of service in each time frame of "t" seconds. A specification of 5 micro-seconds per 20 micro-seconds is a much more real-time QoS than a specification of 5 seconds per 20 seconds, although, on the average, both result in the same amount of time allotted to the task.

The major problem is that the scheduler needs complete knowledge about how long each task is going to take in the near future, and when it will become ready to run. This information is practically impossible to get, and even when it is available, calculation of the optimal scheduling plan is a search problem with high complexity, and hence high cost in time.

Different tasks compete for the same resources: processors, network, memory, disks, ... Much more than in the general purpose OS case, programmers of real-time systems have to take into account *worst-case* scenarios: if various tasks *could* be needing a service, then sooner or later they *will* want it at the same time.

1.3.4. Time data structures

The smallest time slice used in most general purpose operating system is longer than 1 millisecond. Not because the processors are not fast enough to do significant amounts of work in that time slice, but because 32 bit machines have only 2^32 time slices before their timing counter runs over. At 1000 ticks per second, this corresponds to less than 50 days, which is certainly insufficient for servers and embedded systems. Linux uses a scheduling time slice ("jiffie") of 10 milliseconds on most processors. (1 milliseconds on Alpha, which has 64 bit counters.)

The timing constraints of real-time tasks are often expressed with much higher resolutions than those of the general purpose scheduler, i.e., (less than) microseconds instead of milliseconds. Hence, the data structure in which the time is kept should be adapted to this higher rate, in order to avoid overflow. For example, the real-time Linux variants (Chapter 9) use a *high-resolution time data structure* that counts time in *nanoseconds*. A 64 bit integer should do the job in that case, but 32 bits could be too dangerous. (A 32 bit counter overflows after about 4 seconds when counting at a 1 nanosecond rate!) Note that not all compilers can deal with 64 bit integers, such that some assembly coding may be required in some cases.

POSIX has standardized "clocks" and "timers" The *`timespec`* is a data structure that keeps the time in two separate seconds and nanoseconds sub-structures (`include/linux/time.h` of the Linux source tree):

```
typedef long             __kernel_time_t; // include/asm/posix_types.h
typedef __kernel_time_t  time_t;

struct timespec {
   time_t  tv_sec;         /* seconds,  */
   long    tv_nsec;        /* nanoseconds */
};
```

The `timespec` data structure uses 64 bits, but the separation between seconds and nanoseconds is an inefficient way of representing time: there are only approximately 2^30 = 10^9 nanoseconds in one second. So, a little more than two bits of the nanoseconds field are not used. This means that, at each and every addition of a time increment, the software has to check whether the boundary of 1 second hasn't been reached, such that the second field has to be updated. This is more complicated than just having a 64 bit counter that can keep on count without having to check.

1.4. Embedded OS

The concepts introduced in the previous sections apply of course also to embedded operating systems ("EOS"). Embedded operating systems, however, have some features that distinguish them from real-time and general purpose operating systems. But the definition of an "embedded operating system" is probably even more ambiguous than that of an RTOS, and they come in a zillion different forms. But you'll recognize one when you see one, although the boundary between general purpose operating systems and embedded operating systems is not sharp, and is even becoming more blurred all the time.

Embedded systems are being installed in tremendous quantities (an order of magnitude more than desktop PCs!): they control lots of functions in modern cars; they show up in household appliances and toys; they control vital medical instrumentation; they make remote controls and GPS (Global Position Systems) work; they make your portable phones work; etc.

The simplest classification between different kinds of embedded operating systems is as follows:

- *High-end embedded systems.* These systems are often down-sized derivatives of an existing general purpose OS, but with much of the "balast" removed. Linux has given rise to a large set of such derivatives, because of its highly modular structure and the availability of source code. Examples are: routers, switches, personal digital assistants, set top boxes.
- *Deeply embedded OS.* These OSs must be really *very* small, and need only a handful of basic functions. Therefore, they are mostly designed from the ground up for a particular application. Two typical functions deeply embedded systems (used to) lack are high-performance graphical user interfacing or network communication. Examples are: automotive controls, digital cameras, portable phones. But also these systems get more graphics and networking capabilities. . .

The most important features that make an OS into an embedded OS are:

- Small *footprint*. Designers are continuously trying to put more computing power in smaller housings, using cheaper CPUs, with on-board digital and/or analog IO; and they want to integrate these CPUs in all kinds of small objects. A small embedded OS also often uses only a couple of kilobytes of RAM and ROM memory.

- The embedded system should run for years without manual intervention. This means that the hardware *and* the software should never fail. Hence, the system should preferably have no mechanical parts, such as floppy drives or hard disks. Not only because mechanical parts are more sensitive to failures, but they also take up more space, need more energy, take longer to communicate with, and have more complex drivers (e.g., due to motion control of the mechanical parts).
- Many embedded systems have to control devices that can be dangerous if they don't work exactly as designed. Therefore, the status of these devices has to be checked regularly. The embedded computer system itself, however, is one of these critical devices, and has to be checked too! Hence, one often sees *hardware watchdogs* included in embedded systems. These watchdogs are usually retriggerable monostable timers attached to the processor's reset input. The operating system checks within specified intervals whether everything is working as desired, for example by examining the contents of status registers. It then resets the watchdog. So, if the OS doesn't succeed in resetting the timer, that means that the system is not functioning properly and the timer goes off, forcing the processor to reset.

  If something went wrong but the OS is still working (e.g., a memory protection error in one of the tasks) the OS can activate a *software watchdog,* which is nothing else but an interrupt that schedules a service routine to handle the error. One important job of the software watchdog could be to generate a *core dump*, to be used for analysis of what situations led to the crash.

- A long autonomy also implies using as little power as possible: embedded systems often have to live a long time on batteries (e.g., mobile phones), or are part of a larger system with very limited power resources (e.g., satellites).
- If the system does fail despite its designed robustness (e.g., caused by a memory protection fault, Section 1.1.4), there is usually no user around to take the appropriate actions. Hence, the system itself should reboot autonomously, in a "safe" state, and "instantly" if it is supposed to control other critical devices. Compare this to the booting of your desktop computer, which needs a minute or more before it can be used, and always comes up in the same default state…
- It should be as cheap as possible. Embedded systems are often produced in quantities of several thousands or even millions. Decreasing the unit price even a little bit boils down to enormous savings.
- Some embedded systems are not physically reachable anymore after they have been started (e.g., launched satellites) in order to add software updates. However, more and more of them can still be accessed remotely. Therefore, they should support *dynamic linking*: object code that did not exist at the time of start is uploaded to the system, and linked in the running OS without stopping it.

Some applications require all features of embedded *and* real-time operating systems. The best known examples are mobile phones and (speech-operated) handheld computers ("PDA"s): they must be small, consume little power, and yet be able to execute advanced signal processing algorithms, while taking up as little space as possible.

The above-mentioned arguments led embedded OS developers to design systems with the absolute minimum of software and hardware. Roughly speaking, developers of general purpose and real-time operating systems approach their clients with a "*Hey, look how much we can do!*" marketing strategy; while EOS developers say "*Hey, look how little we need to do what you want!*". Hence, embedded systems often come without a memory management unit (MMU), multi-tasking, a networking "stack", or file systems. The extreme is one single monolithic program on the bare processor, thus completely eliminating the need for any operating system at all.

Taking out more and more features of a general purpose operating system makes its footprint smaller and its predictability higher. On the other hand, adding more features to an EOS makes it look like a general purpose OS. Most current RTOS and EOS operating systems are expanding their ranges of application, and cover more of the full “feature spectrum.”

1.5. Operating system standards

Real-time and embedded systems are not a user product in themselves, but serve as platforms on which to build applications. As for any other software platform, the availability of standards facilitates the job of programmers enormously, because it makes it easier, cheaper and faster to develop new applications, and to port an existing application to new hardware. In the world of real-time and embedded systems, standardization is not a burning issue, because many projects in this area have unique requirements, need unique extensions to already existing products, don’t need frequent updates by different people, and are seldom visible to end-users. All these “features” do not really help in forcing developers to use standards… (They do like standard *tools* though, which is one reason for the popularity of the Free Software GNU tools.)

This Section lists some standardization efforts that exist in the real-time and embedded world.

1.5.1. POSIX

POSIX (“Portable Operating Systems Interface”, a name that Richard Stallman came up with) is a standard for the function calls (the *Application Programming Interface*, API) of UNIX-like general purpose operating systems. POSIX has some specifications on real-time primitives too. Its definition of real time is quite loose:

> The ability of the operating system to provide a required level of service in a bounded response time.

The standard is managed by the Portable Application Standards Committee (http://www.pasc.org/) (PASC) of the Institute for Electrical and Electronic Engineers (http://www.ieee.org) (IEEE), and is not freely available. There is an extensive *Rationale* document, that explains the reasons behind the choices that the POSIX committees made, as well as lots of other interesting remarks. That document can be found here (http://www.opengroup.org/onlinepubs/007904975/xrat/contents.html).

The POSIX components relevant to real-time are: 1003.1b (real-time), 1003.1d (additional real-time extensions), 1003.1j (advanced real-time extensions). See this link (http://www.opengroup.org/onlinepubs/007904975/idx/realtime.html) or here (IEEE Std 1003.1-2001) (http://www.unix-systems.org/version3/ieee_std.html) for more details. These standards are often also denoted as ANSI/IEEE Std. 1003.1b, etcetera.

POSIX also defines four so-called *profiles* for real-time systems:

- *PSE51 (Minimal Realtime System Profile)*. This profile offers the basic set of functionality for a single process, deeply embedded system, such as for the unattended control of special I/O devices. Neither user interaction nor a file system (mass storage) is required. The system runs one single POSIX process, that can run multiple POSIX threads. These threads can use POSIX message passing. The process itself can use this message passing to communicate with other PSE5X-conformant systems (e.g., multiple CPUs on a common backplane, each running an independent PSE51 system). The hardware model for this profile assumes a single processor with its memory, but no memory management unit (MMU) or common I/O devices (serial line, ethernet card, etc.) are required.
- *PSE52 (Realtime Controller System Profile)*. This profile is the PSE51 profile, plus support for a file system (possibly implemented as a RAM disk!) and *asynchronous* I/O.
- *PSE53 (Dedicated Realtime System Profile)*. This profile is the PSE51 profile, plus support for multiple processes, but minus the file system support of the PSE52 profile. The hardware can have a memory management unit.
- *PSE54 (Multi-Purpose Realtime System Profile)*. This is the superset of the other profiles and essentially consists of the entire POSIX.1, POSIX.1b, POSIX.1c and.or POSIX.5b standards. Not all processes or threads must be real-time. Interactive user processes are allowed on a PSE54 system, so all of POSIX.2 and POSIX.2a are also included. The hardware model for this profile assumes one or more processors with memory management units, high-speed storage devices, special interfaces, network support, and display devices.

RTLinux claims to comply to the *PSE51* profile; RTAI claims nothing.

Linux's goal is POSIX compliance, but not blindly, and not at all costs. The `/usr/include/unistd.h` header file gives information about which parts of the standard have been implemented already. For example: the implementation of threads (see Chapter 2), and the scheduler modes (see Section 2.7). Many of the real-time POSIX extensions have already been implemented in RTLinux and RTAI (see Chapter 9).

1.5.2. Unix98

UNIX (UNIX98, *Single UNIX Specification, Version 2*) is the standardization of UNIX operating systems driven by the Open Group (http://www.unix-systems.org/unix98.html). It incorporates a lot of the POSIX standards.

1.5.3. EL/IX

EL/IX. The EL/IX (http://sources.redhat.com/elix/) API for embedded systems wants to be a standards-compliant subset of POSIX and ANSI C.

1.5.4. μITRON

μITRON. μITRON (http://www.itron.gr.jp/home-e.html) is a Japanese standard for embedded systems. "TRON" stands for *The Real-time Operating system Nucleus*; the letter "I" stands for *industrial*, and the

“mu” for *micro*. (There are other TRONs too: BTRON for business, CTRON for communication, . . .)

1.5.5. OSEK

OSEK. OSEK (http://www.osek-vdx.org/) is a German standard for an open architecture for distributed vehicle control units. The architecture is open, but no free software implementation is available.

1.5.6. Real-Time Specification for Java

The *Real-Time Specification for Java (RTSJ).* (Java Community Process (http://jcp.org/jsr/detail/1.jsp), rtj.org (http://www.rtj.org/).) is not really an operating system, but a *runtime* for a programming language. The distinction is not really fundamental for normal desktop use; it can be enormous for real-time use, because a runtime must make use of the services of the underlying operating system. That means that a runtime with real-time features is useless on a non real-time operating system.

This specification was released in 2001, and, similar to the POSIX specifications, it is *not* an implementation; some commercial implementations are already available. The basis prescriptions of the specification are:

- Implementations of the specification are allowed to introduce their own optimizations and extensions, such as, for example, scheduling algorithms or garbage collection.
- The *minimum* task management includes static priority-based preemptive scheduling, with at least 28 priority levels.
- Priority inversion “prevention” (Section 4.9) is mandatory.
- An implementation must include classes that provide an asynchronous event mechanism.
- Exceptions must be allowed to change the context to another thread.
- Clases must be provided to allow direct access to physical memory.

1.5.7. Ada 95

Ada 95 real-time specifications.

The MaRTE OS (http://marte.unican.es/) is an example of a free software real-time kernel for embedded applications that complies with Minimal Real-Time POSIX.13. Most of its code is written in Ada with some C and assembler parts. The Ada runtime from the GNU Ada Toolkit (GNAT) (ftp://ftp.cs.nyu.edu/pub/gnat) has been adapted to run on the kernel. The Ada compiler comes under the GPL, but the runtime has a modified GPL license that allows it to be used without constraints in commercial systems.

OpenRavenscar (http://polaris.dit.upm.es/~ork/) is another free software real-time kernel in Ada.

1.5.8. Real-Time CORBA

The *Open Management Group (OMG)* has released a specification of a "real-time" component broker interface, called *Real-Time CORBA* ("RT-CORBA"). This is *not* a piece of software, but a *specification* interface. So, various implementations can satisfy the interface, with very different real-time behaviour. The RT-CORBA specifications allow the component builder to specify some *desired* properties that are common for real-time tasks, such as static priority levels or time-outs. These specifications have to be mapped onto real (RT)OS primitives by the specific implementation(s) used in the application.

1.6. Linux for real-time and embedded

Linux is a *general purpose* operating system, with a non-pre-emptable kernel: it wants to give all tasks a *fair* share of the resources (processor, memory, peripheral devices, ...), and it doesn't interrupt kernel activities. Linux's basic user space scheduler is of the *time slicing* type: it gives more or less equal time slices to different tasks. It is possible to change the priorities of user space tasks to some extent (using the `nice` command), but not enough to make the scheduling deterministic. Other reasons why Linux is a poor RTOS are the unpredictable delays caused by non-pre-emptable operations running in kernel space, and by the mere size of that kernel. Indeed, *nobody* can understand the kernel sufficiently well to be able to predict how long a certain operation is going to take.

All remarks above hold for all general purpose operating systems, such as Windows, AIX, IRIX, HP-UX, Solaris, etc. It may sound strange at first, but "good old" DOS was much closer to being an RTOS than Linux, because its scheduler was much less "fair" and advanced, and it had fewer system services to look after. (However, DOS is only an advantage if there is only *one* real-time task!) Because none of the desktop or server operating systems is a good candidate for real-time and/or embedded applications, several companies have started to develop special purpose operating systems, often for quite small markets. Many of them are UNIX-like, but they are not mutually compatible. The market is very fragmented, with several dozens of RTOSs, none of which holds a majority of the market. At least, this was the case *before* Linux appeared on the radar of real-time and embedded system companies. Since about the year 2000, the market has seen lots of mergers and acquisitions, and substantial efforts from the established RTOS companies to become as "Linux-compliant" as possible.

The fact that Microsoft tries to enter the market too (with its PocketPC/Windows CE product line) is only accelerating this evolution. History has learned that the fragmented UNIX desktop and server markets were easy targets for Microsoft..., even with inferior technology. So, hopefully the competitors have learned from this experience.

While the Linux kernel people, headed by Linus Torvalds, are very keen on making the general support and performance of Linux better, their interest in real time is very small, to say the least... No efforts to make Linux into a real RTOS have to be expected from that side, but the kernel *is* evolving towards higher pre-emptability (first of all, because this is necessary if one wants to scale Linux to more than, say, two CPUs).

Torvalds has mentioned two reasons why he doesn't want to make Linux into a real-time operating system:

- Computers are getting faster all the time, such that a general-purpose operating system will satisfy the requirements of more and more "real-time" users. (That is, those that require a *fast* system, which is not the same as a *deterministic* system.)
- Offering hard real-time features in a general-purpose OS will quickly result in "bad behaviour" of application programmers (http://kernelnotes.org/lnxlists/linux-kernel/lk_0006_05/msg00180.html): they will all want their application to perform best, and program it with high priority. Experience has shown many times that this leads to incompatible timing constraints between different applications rather sooner than later.

However, there are no technical reasons why Linux would not be able to become (more of) an RTOS, and much technology to make Linux more powerful on the high-end server systems is also useful for real-time and embedded purposes: real multi-threading in the kernel, finer locks and scheduling points needed for SMP systems, migration of processes over CPUs, "hot-swapable" devices, etc.

Anyway, quite a lot of Free Software efforts have started to contribute software in the area of real-time and embedded systems. These contributions can be classified as follows:

- *Eliminating functionalities from the standard Linux kernel.*

  This approach aims at reducing the memory footprint of the operating system, and is hence mainly focused on embedded systems. uCLinux (http://www.uclinux.org) is an example. Other projects develop small and simple C libraries, because the current versions of the GNU tools have become quite large; for example, BusyBox (http://www.busybox.net) (a replacement for most of the utilities one usually finds in the GNU fileutils, shellutils, etc.); μclibc (http://www.uclibc.org) (a small version of the general C library).

- *Patches to the standard Linux kernel.*

  This approach replaces the standard scheduler of Linux with a more deterministic scheduling algorithm, and adds scheduling points to the Linux source tree, in order to make the kernel more responsive.

- *Real-time patches underneath the Linux kernel.*

  This approach runs Linux as a low-priority process in a small real-time kernel. This kernel takes over the real hardware from Linux, and replaces it with a software simulation.

  The two major examples that follow this road are RTLinux (http://www.rtlinux.org/) (Section 9.2) and RTAI (http://www.rtai.org/) (Section 9.3).

- *Linux-independent operating systems.*

  These projects have been developed completely independently from Linux, and some of them are even older. Some examples are RT-EMS (http://www.rtems.com/), and eCos (http://sources.redhat.com/ecos/).

The Linux kernel that supports a typical desktop computer is several hundreds of kilobytes large. And that does *not* include the memory taken up by the Linux tools and the users' applications. Hence, the Linux footprint is too large for many embedded systems. It also takes about a minute or so to boot a PC-like computer, which is much too long for most embedded systems. And it expects a hard disk to work with, and a power supply of more than 100 Watts for modern high-end CPUs, video cards and hard disks.

However, one of the nicest things about Linux is its enormous configurability, of which you can get a taste if you compile your own Linux kernel. That kernel can be constructed out of lots of more or less independent modules, and you just leave out the modules that are not needed in your system. If your application requires no ethernet card, leave out the network drivers; if you don't need a screen, why bother with installing the X Window System; etc. This means that many people have configured Linux-derived systems that become so small that they fit on a single floppy.

The previous paragraphs may suggest that Linux proper has no chance at all at being used as an embedded OS. However, it has some advantages that may turn out to be decisive in the not too distant future (certainly because memory and CPUs become cheaper): its configurability, its ability to be administered from a distance, *and* its many ways to add security features.

Chapter 2. Task management and scheduling

This Chapter explains what *task management* means, and how it can influence the real-time behaviour of an operating system. Concrete examples come from the POSIX standard, but the concepts are identical for other task management APIs. *Scheduling* of tasks is one of the responsibilities of the task management with influence on the real-time behaviour of the system. Other responsibilities are: task creation and deletion, linking tasks to interrupts and deferred interrupt servicing, and assignment and change of scheduling priorities.

2.1. Processes and threads

We use “task” as the generic name for both *processes* and *threads*. A process is the normal “unit of execution” in UNIX systems: if you compile a C program that has one single `main()`, then running this program requires one process. (That process can generate itself other processes too, of course.) The operating system must provide several services to each process: memory pages (in virtual memory and in physical RAM) for code, data, stack and heap, and for file and other descriptors; registers in the CPU; queues for scheduling; signals and IPC; etc.

A process can spawn new processes (“children”), either by starting up an independent process via a system call, or by `fork`-ing itself. (The Linux kernel uses a somewhat other approach, with the `clone()` function, see Section 2.3.) The forked process is a copy of the parent process, but it gets its own memory, registers, file descriptors, and process identifier. Starting a new process is a relatively heavy task for the operating system, because memory has to be allocated, and lots of data structures and code segments must be copied.

A thread is a “lightweight” process, in the sense that different threads share the same address space. That is, they share global and “`static`” variables, file descriptors, signal bookkeeping, code area, and heap, but they have their own thread status, program counter, registers, signal mask (in Linux but not in UNIX), and stack. The interesting fact from an RTOS point of view is that threads have shorter creation and *context switch* times, and faster IPC (see Chapter 4). A “context switch” is the saving of the state of the currently running task (registers, stack pointer, instruction pointer, etc.), and the restoring of the state of the new task. Other advantages for using multiple threads within a process are:

- The threads can be run on separate processors.
- The tasks can be prioritized, so that a less important computation can, in response to an external event, be suspended to process that event.
- Computation can occur in one thread, while waiting for an event, such as the completion of I/O, can be outsourced to another thread.

On the other hand, using threads requires functions to be made “thread-safe”: when a function `func()` is called in one thread, this thread can be pre-empted by another thread, which, in turn, can call the same function; hence, this function should not keep intermediate data in variables that are shared between the different threads.

Many modern CPUs offer functionality such as floating-point calculation, digital signal processing (e.g., “MMX”), or on-chip memory caches. These functions require extra registers and/or operations, so, when this extra functionality can be avoided, real-time determinism is increased (because the context switch time is lower if less registers have to be saved and restored). For example, Linux doesn’t save the floating point registers for kernel tasks and interrupt service routines.

2.2. POSIX thread management

The POSIX operating system standard (see Section 1.5) has an extensive threads API, which all UNIX-like operating systems implement (albeit to varying degrees). The thread implementation in Linux is not the most complete The real-time operating systems discussed in later Chapters all have decent, but not complete, POSIX thread functionality. The reasons why many operating systems don’t implement the full POSIX standards are: (i) POSIX is not a single, rigid standard, but a large set of complementary standards with different focus points; (ii) one doesn’t need the whole API to build functional and efficient software systems; (iii) some parts of the standard require complicated implementations with meager practical and not-unique advantages; (iv) some features made it into the standard for the sole purpose of being backwards compatible with older existing UNIX systems.

The POSIX API provides the following function calls (and others!) for thread creation and deletion:

```
int pthread_create(
  pthread_t *thread,                // thread data structure
  pthread_attr_t *attr,             // attributes data structure
  void *(*start_routine) (void *), // function to execute
  void *arg                         // argument to pass to function
);
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **status),
int pthread_detach(pthread_t thread),
int pthread_cancel(),
```

(The initial letter “`p`” indicates the POSIX heritance.) Thread creation involves some overhead, because memory has to be allocated for the new thread; in a real-time setting, the memory also has to be *locked* into RAM, in order to be sure that no time will ever be lost because the memory pages have to be swapped in from disk when needed. Similarly, freeing memory at thread deletion is also an overhead. So, a real-time application should do thread creation and deletion *outside* of the real-time activity.

Other overhead caused by task management is: satisfying requested changes in the timing or priority properties of tasks, and the maintenance of the task queues at all priority levels when tasks are woken up, put asleep, made running, obliged to wait for a blocking IPC call, etc.

In the `pthread_create()`, the programmer can specify the *run-time priority* of the task, as well as the scheduling policy to use, through the `pthread_setschedparam()` function call.

`pthread_join()`, `pthread_detach()`, and `pthread_cancel()` are different ways to end the execution of a task. A task should indeed not be deleted blindly, because it shares a lot of its components with other tasks, so its memory space and locks should not be released when its cousin tasks are still using them. Especially *cancelling* a thread from within another thread is dangerous: it practically impossible to tell what resources the cancelled task is holding (including locks!). So, POSIX prescribes a procedure to cancel tasks. `pthread_cancel()` does not cancel the task immediately, but is only a *request* to the operating system to cancel the task. How the task is cancelled depends on how the task initialised its own cancellation policy, via:

- `int pthread_setcancelstate(int state, int *oldstate)`: atomically sets the calling task's cancellability *state* to the indicated *`state`* and returns the previous cancellability state in *`oldstate`*. Possible values for *`state`* are *`PTHREAD_CANCEL_ENABLE`* and *`PTHREAD_CANCEL_DISABLE`*.
- `int pthread_setcanceltype(int type, int *oldtype)`: atomically sets the calling task's cancellability *type* to the indicated *`type`* and returns the previous cancellability type in *`oldtype`*. Possible values for *`type`* are *`PTHREAD_CANCEL_DEFERRED`* and *`PTHREAD_CANCEL_ASYNCHRONOUS`*.

The default cancellation type and state are *`PTHREAD_CANCEL_DEFERRED`* and *`PTHREAD_CANCEL_ENABLE`*.

Cancellation happens immediately if the task has chosen the *`PTHREAD_CANCEL_ASYNCHRONOUS`* policy; so, this policy should only be chosen when the programmer is certain that the task can be killed at any time, without compromising the rest of the system. If the task has chosen the *`PTHREAD_CANCEL_DEFERRED`* policy, it is cancelled only when it reaches a so-called *cancellation point*. These OS-dependent points are function calls where the task tests whether it has received a cancellation request. (Or rather, the operating system does the test for it, as well as the cancellation handling, discussed below.) Cancellation function calls are typically calls that might block for a long time, such that the OS need only check for pending cancellation requests when the operation is about to block indefinitely. This includes, but is not at all limited to, `pthread_cond_wait()`, `pthread_cond_timedwait(()`, or `sem_wait()`, or `sigwait()`.

The task that one wants to cancel can postpone cancellation in order to perform application-specific cleanup processing. It does this by "pushing" cancellation *cleanup handlers* every time that it acquires some resource. As the task leaves the last cancellation point before releasing a resource, it needs to "pop" the cleanup handler it had pushed earlier for this resource. Pushing and popping is done by the `pthread_cleanup_push()` and `pthread_cleanup_pop()` function calls. Every cleanup handler that is still on the cleanup stack is invoked (in *Last-in, First-Out* order) when the task is cancelled, and its job is to cleanly release the resource. The task terminates when the last cleanup handler returns. The task exit status returned by `pthread_join()` on a cancelled task is *`PTHREAD_CANCELED`*.

(This behaviour is quite standard in many software tasks; Section 15.4 gives the generic software design behind such behaviour.)

The cancellation procedures above might seem a bit involved, but that's due to the complexity of the problem one wants to solve: making sure that another task exits without blocking other tasks. Anyway, this kind of cancellation should be avoided whenever possible. The clean solution is to let all tasks in your application react to a *condition variable* that indicates that it must shut down itself (Section 15.4.4).

An RTOS must also allow to specify the *timing* with which threads have to run. One typically uses two timing modes:

- *Periodic:* the task must run at regular intervals.
- *One-shot:* the task must run only once, at a predefined instant in time.

One-shot timing sometimes requires a bit more overhead, because of a more involved hardware timer programming. POSIX has no standardized function calls for *periodic* timing. The reasons are that: (i) there are multiple ways in which the desired functionality can be programmed with already existing POSIX primitives; and (ii) most applications have to break the periodic loop in one way or another anyhow, depending on application-specific conditions. Because of the lack of a (POSIX) standard API for periodic thread timing, different operating systems implemented the functions on their own, such that application programs will most probably have portability problems in this area. For example, RTLinux uses `pthread_make_periodic_np()` for both options (the `_np` suffix stands for “non-portable”), while RTAI has `rt_set_periodic_mode()` and `rt_set_oneshot_mode()`.

As examples of alternatives for the periodic timing function, POSIX provides the `usleep()` and `nanosleep()` function calls. These put tasks asleep with a high timing resolution (microsecond, respectively nanoseconds). The achievable resolution depends of course on the type of CPU.

Some other often-used functionality that POSIX has not standardized is: to allow the use of floating point operations in a thread (for which, e.g., RTLinux has introduced `pthread_setfp_np()`); to suspend execution of *another* thread than the one that executes the function (“`pthread_suspend_np(another_thread)`”); and “`pthread_wakeup_np(another_thread)`” to resume execution of the other thread. Note again the “`..._np`” suffix.

The floating point selection option was considered too low level and hardware dependent to put into the POSIX standard. Saving a couple of registers more or less is more of a matter of *optimization*, and such things don’t belong in a standard. The Linux scheduler, for example, always saves floating point registers of *user space processes* by default.

The `pthread_suspend_np()` and `pthread_wakeup_np()` functions are *dangerous* (see below), and the POSIX committee had very good reasons not to include them in the standard. However, many users think they are “user-friendly”, because they sometimes save them a lot of keystrokes. The danger of `pthread_suspend_np()` is that, while its use is convenient to stop a thread, it leaves that thread most probably in an undefined state, such that it’s hard to predict what the thread is going to do when `pthread_wakeup_np()` starts it again!

The proper way of suspending the execution of a thread is to let the thread do it *itself*, at a moment it is ready to do so, i.e., it is in a well-defined state, from which it can restart in a deterministic way. Chapter 17 gives some more detailed examples.

2.3. Linux tasks and tasklets

The above-mentioned distinction between “process” and “thread” is not what Linus Torvalds has in mind. He thinks the really important concept is the *Context of execution*: that includes things like CPU state (registers, etc.), memory management state (page mappings), permission state (user ID, group ID), code to execute, and various “communication states” (open files, signal handlers, etc.). An email by Torvalds in which he explains his (and hence Linux’s) point of view can be found here (http://www.uwsg.iu.edu/hypermail/linux/kernel/9608/0191.html). POSIX threads are offered on Linux as a *library*, and basically only because of compliance with the standard. Anyway, they are just one single possible way to share context. And the Linux kernel offers a more flexible alternative: the `clone()` creates a new “task”, with a large choice in what parts of the *context of execution* one wants to share between the new task and the task that creates it. See the corresponding man page for more details.

Many operating systems provide another primitive besides threads or processes, that programmers can use to execute functionality. Linux and RTAI call it *tasklets*, Section 2.6. A tasklet is a *function* whose execution can be asked for by any kernel task, and that the operating system will execute *before* it does its next scheduling. At that moment, the OS executes these functions one by one. So, the important features of tasklets are:

- They are a more “lightweight” primitive than tasks, to execute functions outside of, and prior to, the normal scheduling. of tasks.
- They are not pre-empted by normal tasks.

But tasklets *can* be pre-empted by interrupts, because the kernel has enabled all hardware interrupts when it runs the tasklets. Tasklets are typically only executed once, but some operating systems (e.g., RTAI) offer periodic execution of tasklets, by registering them with a *timer*. The tasklet primitive is also very useful as a so-called *Deferred Service Routine (DSR)*, Section 3.4.

2.4. Scheduling

Some texts make a distinction between *scheduling* and *dispatching*, with dispatching being the simplest of the two operations:

- *Scheduling*: determining the order and the timing (i.e., the “schedule”) with which tasks should be run.
- *Dispatching*: the dispatcher starts and stops the tasks, i.e., it *implements* the schedule.

This text only uses the term “scheduling”.

A primary responsibility of an RTOS is to make sure that all tasks meet their *timing* constraints. Timing constraints come in different flavours (deadline, zero execution time, QoS), and for every task the constraints can change over time. For example, a motion generator for a mobile robot has much more constraints to take into account when it navigates in an environment with many nearby obstacles, while its job is much easier in open areas. Or, users of a multimedia server have different QoS requirements for editing one video stream than for the editing and synchronization of several streams.

So, a “one-size-fits-all” scheduling algorithm does not exist. Although that is exactly what a general purpose operating system hopes to offer. Hence, it should come as no surprise that there is a vast literature on the theory of scheduling, accompanied by a large variety of (un)implemented scheduling algorithms. A theoretically optimal schedule can only be reached in the unlikely situation of *complete knowledge* about the processing, synchronization and communication requirements of each task, and the processing and timing properties of the hardware. This state of complete knowledge is seldom reached in real-world applications, especially when the requirements are *dynamic* (i.e., time varying). And even with complete predictability, the general scheduling problem is *NP-complete*, which means that its complexity increases exponentially with the number of tasks and constraints involved in the scheduling. And hence, the scheduling algorithms don’t scale well under a growing load and/or hardware resources. This does *not* imply, however, that the problem is infeasible for applications with only a few, well-defined tasks.

Each OS has a scheduler *function* (let’s call it `schedule()`), that implements the scheduling algorithm. (Later sections discuss the most common scheduling algorithms.) This scheduler is *not* a task in itself: it is a function call, that is called at various points in the kernel. These points are, not surprisingly, called *scheduling points*. Typical scheduling points are: end of interrupt service routines (Section 3.3), the moments when tasks want to go to sleep for one reason or another, or when they become ready to run.

Scheduling is pure overhead: all time spent on calculating which task to run next is lost for the really productive tasks. And trying to use more optimal schedulers isn’t always a clever “solution”: advanced schedulers consume (often unpredictably!) more time and resources, and their increased complexity makes it more difficult for programmers to work with. Hence, the chance that those programmers make the wrong design decisions increases. Simplicity is especially a key feature for real-time and embedded systems; complex schedulers appear more in Operations Research applications, where the scheduling problem and its algorithmic complexity are comparable to the operating system case, but where the real-time constraints and the predictability of the cost of tasks are more manageable.

(TODO: explain (POSIX) cancellation points: why are they needed? what makes a point a valid cancellation point? Warn against using cancellation, because it’s so error prone. Not only from the OS point of view (that OS must make sure its thread and lock bookkeeping remains consistent, which is not a simple job), but also from the application point of view (how do you make sure that there is no race between one thread trying to cancel another thread, and a third thread that still wants to interact with that to-be-cancelled thread? It’s way better to have each thread exit itself explicitly, and to have an explicit exit condition for each thread. And to make thread interaction *asynchronous*.)

2.5. Priority-based scheduling

The *simplest* approach to the scheduling problem is to assign *static priorities* to all tasks. That means that the priority is given to the task at the time it is created. The scheduler function `schedule()` is then very simple, because it looks at all wait queus at each priority level, and starts the task with the highest priority that is ready to run.

Using priorities implies using *pre-emption*: `schedule()` interrupts a lower priority task in order to run a higher priority task that requests it. Pre-emption means that the running task’s context is switched out, and the new task's context is switched in.

One classifies priorities into *statically* and *dynamically* assigned priorities. In the former case, a task is given a priority by the programmer at design time (or by the operator at system initialization time), and it keeps this priority during its whole lifetime. In the dynamic case, `schedule()` becomes more complex, because it has to calculate the task's priority on-line, based on a number of dynamically changing parameters (time till next deadline; amount of work to process; etc.). As described before, the optimal solution to a scheduling problem is usually impossible to find, so scheduling is often based on a set of *heuristics*. This is the case for real-time as well as non-real-time schedulers. The heuristics in a general purpose OS can be quite involved, but real-time and embedded operating systems mostly use simple heuristics. Because "simple" means: faster and smaller and more predictable! Examples of such simple dynamic scheduling algorithms, that are sometimes used to replace static priority scheduling, are:

- *Rate monotonic* (RM). A task gets a higher priority if it has to run more frequently. This is a common approach in the case that *all tasks are periodic*. So, a task that has to run every n milliseconds gets a higher priority than a task that runs every m milliseconds when n<m. Hence, changing the scheduling frequency of a task on-line also changes its priority. The scheduler needs to know the periods of all tasks it has to schedule.
- *Earliest deadline first* (EDF). At each instant in time, there are a number of tasks that need to be finished in the near future. A task with a closer deadline gets a higher scheduling priority. The scheduler needs not only to know the deadline time of all tasks it has to schedule, but also their duration.

If different tasks in the system request different scheduling policies, the operating system has to make trade-offs in determining the relative "weight" to give to each of the scheduling algorihtms. These trade-offs will most probably be quite arbitrary, so porting your application between operating systems could lead to different scheduling results.

Priority-based scheduling is simple to *implement*, because `schedule()` just has to look at the tasks in the highest priority queue that are ready to be scheduled, and to start the first one in this queue. Priority-based scheduling, however, is *difficult for the application programmers*: they must try to map the often complex ("high-dimensional") synchronization interdependencies between the different threads in their application onto the *linear scale* offered by priorities! One often-observed phenomenon in real-time applications that grow over time, is that the programmers tend to raise the priorities of some threads, every time they notice that the introduction of new functionality (and hence new threads) disturbs the synchronization of the existing threads. Chapter 14 gives some more examples of the negative effects of "coupling", and Chapter 15 discusses time-proven approaches to take care of complex interdependencies.

So, the problem with priority-based scheduling is that it is an *indirect* way to specify how to cope with timing and synchronization constraints: at run-time, `schedule()` doesn't take these constraints themselves into account, but knows only about the priorities, which are the programmer's indirect model of the constraints.

In practice, all RTOSs at least offer static priority-based scheduling. Many also implement other algorithms. Not always because of the intrinsic added value of the algorithm, but rather because of typical marketing drives: users tend to buy software products with the highest number of features, even if they risk to drown in the complexity and "feature bloat" (whose implications they often even don't understand. . .). One of the more serious feature bloat examples in priority-based scheduling is the so-called *priority inheritance* "solution" to the *priority inversion* phenomenon (see Section 4.8), that occurs when tasks share resources which they should not access concurrently.

2.6. Priority spaces

Many operating systems (especially RTOSs) let all tasks (system tasks as well as user tasks) live in the same priority space: any task can be given any priority within this space. Others, such as Linux, UNIX, or Microsoft NT, have separate priority spaces for different kinds of tasks. Linux has two: *user space* and *kernel space*. Tasks running in user space can change their priorities (through the `nice()` function call), but all of them are pre-empted by any task in kernel space. Kernel space itself has three priority levels:

1. *Interrupts*: the "task" that services a hardware interrupt (timer, network, keyboard, . . .) has the highest priority. Such a task is called an *interrupt service routine (ISR)*. (Section 3.3.) It should be as short as possible, because it runs with all other interrupts disabled. An ISR is not really a task, but just a *function call*, and its execution is not determined by the scheduler: the ISR is executed immediately at the occurrence of an hardware interrupt, by the hardware of the interrupt controller and the CPU (Section 3.2). The operating system software is not involved at all.
2. *Tasklet* functions (Linux specific, Section 3.4) and *Deferred Service Routines* (terminology often used outside of Linux) are *functions* (again, *not* tasks!) that run at the second highest priority. Only an hardware interrupt can pre-empt them. A tasklet can be activated by any kernel task; a deferred interrupt function (Section 3.4) is typically triggered by a hardware interrupt service routine, to further process an interrupt after the ISR has finished. Both have the same properties, and are executed after all hardware interrupt service routine have finished, and before the "normal" tasks are scheduled; interrupts are enabled when they run. In contrast to the hardware interrupts, the operating system software *is* involved in determining when they are executed.
3. *All other kernel tasks* run at the lowest priority level in the kernel. They pre-empt every user space task.

There is no consensus about the relative merits of having separate user and kernel spaces: some consider it to be a design advantage ("divide et impera"), while others experience it as an unnecessarily artificial constraint on their flexibility.

2.7. Linux scheduler

The scheduler implemented in the file `/usr/src/linux/kernel/sched.c` of the Linux source tree works with three scheduling modes (which are defined in the POSIX standard): *`SCHED_RR`*, *`SCHED_FIFO`* and *`SCHED_OTHER`*. *`SCHED_OTHER`* is the default. The scheduling mode of a task is set by the POSIX `sched_setscheduler()` system call.

`SCHED_RR` is the *round-robin* time slicing algorithm. After a task finishes its time slice, it is moved to the tail of its priority queue, such that another task in the same priority level can start running. If there is no other task at this priority, the pre-empted task can continue.

`SCHED_FIFO` is a *First-In, First-Out* scheduling algorithm: the tasks in one priority level are scheduled in the order they get ready to run; once a task is scheduled, it keeps the processor until pre-empted by a higher priority task, until it releases the processor voluntarily, or until it has to wait to get access to some resource. This scheduler mode is often called “POSIX soft real-time” because it corresponds to the most common real-time scheduling approach with static priorities, but without the other necessary real-time components.

The behaviour of the *`SCHED_OTHER`* scheduler function is not prescribed by the POSIX standard. It is meant to give freedom to the operating system programmers to implement their own scheduling algorithm. In Linux, as in all general-purpose operating systems, the *`SCHED_OTHER`* scheduler function tries to combine two conflicting performance measures: maximimum throughput and good response to interactive users. The Linux scheduler calculates a “goodness” value for each candidate task, based on a number of *heuristic rules*. Recently, the scheduler function got a lot of attention from the Linux kernel developers, since a new *O(1)* (“order one”) scheduling algorithm was introduced. *O(1)* means that the function’s computational time does not increase with the number of tasks that must be scheduled. This has led to a more responsive kernel, certainly in combination with the increased number of pre-emption points (Section 2.8), which all lead to a call to the scheduler function.

Kernel tasks with the *`SCHED_OTHER`* scheduling policy receive the lowest priority, “0”, while the *`SCHED_RR`* and *`SCHED_FIFO`* policies can use priority levels from “1” to “99”. User space tasks are always scheduled with the *`SCHED_OTHER`* policy. The priority levels 0 to 99 are prescribe in the POSIX standard, and the portable POSIX way to find out about the minimum and maximum scheduling priorities is through the `sched_get_priority_min()` and `sched_get_priority_max()` system calls. Both take one of the priority policies as their argument.

The scheduling for *Symmetric Multi-Processor* (SMP) systems is basically the same as for the uni-processor case. There are some extra function calls to assign a task or an interrupt to a specific processor, if the programmers desires so. This decision could lead to more efficient execution, because it increases the chance that the task’s or ISR’s code can permanently be kept in the cache of that particular processor.

2.8. Linux real-time scheduling

Linux will not become a full-fledged RTOS, for the simple reason that the requirements for a general-purpose operating system are very different from those of an RTOS. However, soft real-time additions to the standard Linux kernel have been developed in several places.

One active source of soft real-time efforts has been the audio and video community: in this area, Linux and Microsoft NT perform poorly, in comparison to, for example, BeOS and Microsoft Windows 95. The reason is that Linux and Microsoft NT can’t guarantee these multi-media tasks a deterministic share of the resources (QoS). BeOS does offer QoS scheduling, while Microsoft Windows 95 simply has much less things to do than a “real” operating system. . . .

Another reason for soft real-time work is the drive to make Linux scale better on multi-processor systems. In this context, it is important to keep the locks on kernel functionality as small as possible, because if one processor needs a lock, the other processors are also disturbed in their activity. The expectation is that the scalability activity will make Linux into an operating system that can almost guarantee milli-second deadlines (i.e., "soft real time"), without making it into a real RTOS.

Here is a (non-exhaustive) list of efforts to improve on latency problems in the Linux kernel:

- Montavista's Hard Hat Linux (http://www.mvista.com/products/hhl.html) with its *pre-emption (http://www.mvista.com/dswp/PreemptibleLinux.pdf)* patches. These are currently maintained by Robert Love (http://www.tech9.net/rml/linux/), and gradually introduced in the new 2.5.x kernels. The idea is to see whether the scheduler could run, at the moment of that a kernel spinlock (Section 4.6.3) is released, or an interrupt routine (Chapter 3) exits. Commands exist to disable or enable kernel pre-emption.
- Ingo Molnar's *low-latency* patches, now maintained by Andrew Morton (http://www.zipworld.com.au/~akpm/). They introduce more scheduling points (http://www.linuxdevices.com/articles/AT8906594941.html) in the kernel code, such that the time is reduced between the occurrence of an event that requires rescheduling and the actual rescheduling. Probably, this work will be combined with the pre-emption work mentioned above.
- TimeSys Linux/RT (http://www.timesys.com) develops and commercializes the so-called "Resource Kernel" loadable module, that makes the standard Linux kernel pre-emptable, and that allows to build QoS scheduling for user tasks.
- KURT (http://www.ittc.ukans.edu/kurt/) (*Kansas University Real-Time Linux*). KURT Linux allows for explicit scheduling of any real-time *events* rather than just *tasks*. This provides a more generic framework onto which normal real-time process scheduling is mapped. Since event scheduling is handled by the system, addition of new events such as periodic sampling data acquisition cards (video, lab equipment, etc.) is highly simplified. KURT introduces two modes of operation: the normal mode and the real-time mode. In normal mode, the system acts as a generic Linux system. When the kernel is running in real-time mode, it only executes real-time processes. While in real-time mode, the system can no longer be used as a generic workstation, as all of its resources are dedicated to executing its real-time responsibilities as accurately as possible.
- The LinuxBIOS (http://www.linuxbios.org) project allows to get rid of the usual *BIOS chips that manage part of the hardware, and that introduce significant delays when booting. A LinuxBIOS startup can take place in a few seconds, booting immediate in a ready-to-go kernel.*
- Linux-SRT (http://www.uk.research.att.com/~dmi/linux-srt/) is a QoS scheduler.
- QLinux (http://www.cs.umass.edu/~lass/software/qlinux/) is a QoS scheduler.
- Fairsched (http://fairsched.sourceforge.net/) is a hierarchical QoS scheduler: tasks are divided into groups and each *group* receives guaranteed CPU time allocation proportional to its weight. The standard scheduler is used to schedule processes within a group.
- DWCS (http://www.cc.gatech.edu/~west/dwcs.html) (*Dynamic Window-Constrained Scheduling*) is a QoS scheduler, parameterizing the service in terms of a *request period* and a *window constraint*. The request period is the time interval over which a task must receive some share of the CPU; the window constraint is the value of that minimum share the task much receive during this "window."

Chapter 3. Interrupts

This Chapter explains the basics of interrupt servicing in a computer system, with again an emphasis on the real-time application. Interrupt hardware and software come in a great variety of implementations and functionalities, so some of the concepts talked about in this Chapter may not be relevant to your system.

3.1. Introduction

Interrupts are indispensable in most computer systems with real-time ambitions. Interrupts have to be processed by a so-called ISR (*Interrupt Service Routine*). The faster this ISR can do its job, the better the real-time performance of the RTOS, because other tasks are delayed less. Timers are one example of peripheral devices that generate interrupts; other such devices are the keyboard, DAQ (*Digital AcQuisition*) cards, video cards, the serial and parallel ports, etc. Also the processor itself can generate interrupts, e.g., to switch to the "protected mode" of the processor, when executing an illegal operation, as part of a debugging session, or when an "exception" is raised by an application program.

3.2. Interrupt hardware

An interrupt-driven system (which many RTOSs and EOSs are) typically has one or more of the following hardware components:

- *Interrupt vector*. Many systems have more than one hardware interrupt line (also called *interrupt request (IRQ)*, and the hardware manufacturer typically assembles all these interrupt lines in an "interrupt vector". The INTEL 80x86 processors' interrupt vector contains 256 entries, and is called the *Interrupt Description Table (IDT)*, [Hyde97]. (But most PCs manufacturers make only 16 of these interrupts available as *hardware* interrupts! See below.) The interrupt vector is an array of pointers to the interrupt service routines (Section 3.3) that will be triggered when the corresponding interrupt occurs. The vector also contains a bit for each interrupt line that signals whether there is an interrupt *pending* on that line, i.e., a peripheral device has raised the interrupt, and is waiting to be serviced.

  Some processors use *non-vectored* interrupt processing: when an interrupt occurs, control is transfered to one single routine, that has to decide what to do with the interrupt. The same strategy is also used, in software, in most operating systems to allow multiple devices to share the same interrupt.

- *Synchronous or software interrupt*. A synchronous interrupt (also called a software interrupt , or a trap) is an interrupt that is not caused by an (asynchronous) hardware event, but by a specific (synchronous) *machine language operation code*. Such as, for example the `trap` in the Motorola 68000, the `swi` in the ARM, the `int` in the Intel 80x86, by a divide by zero, a memory segmentation fault, etc. Since this feature is supported in the *hardware*, one can expect a large number of different, not standardized, names and functions for software interrupts...

Major differences between asynchronous/hardware interrupts and synchronous/software interrupts, on most hardware, is that:

1. Further interrupts are *disabled* as soon as an hardware interrupt comes in, but not disabled in the case of a software interrupt.
2. The handler of a software interrupt runs in the context of the interrupting task; the ISR of an hardware interrupt has not connected task context to run in. So, the OS provides a context (that *can* be the context of the task that happened to be running at the time of the interrupt).

Hardware and software interrupts do share the same interrupt vector, but that vector then provides separate ranges for hardware and software interrupts.

- *Edge-triggered and level-triggered interrupts.* From a hardware point of view, peripheral devices can transmit their interrupt signals in basically two different ways:
  - *Edge-triggered*. An interrupt is sent when the interrupt line changes from low to high, or vice versa. That is a almost "zero time" event, which increases the chances for a *hardware* loss of interrupts by the interrupt controller. Moreover, if multiple devices are connected to the same interrupt line, the operating system *must* call all registered interrupt service routines (see Section 3.3), because otherwise it could cause a *software* loss of an interrupt: even if it detected only one edge transition, and its first ISR acknowledged the receipt of this interrupt, it could still be that it missed another edge transition, so it can only be sure after it has given all ISRs the chance to work. But of course, this is not an efficient situation.
  - *Level-triggered*. An interrupt is signaled by a change in the *level* on the hardware interrupt line. This not only lowers the chance of missing a transition, but it also allows a more efficient servicing of the interrupts: each ISR that has serviced the interrupt will acknowledge its peripheral device, which will take away its contribution to the interrupt line. So, the level will change again after the last peripheral device has been serviced. And the operating system should not try all ISRs connected to the same hardware interrupt line.
- *Interrupt controller.* This is a piece of hardware that shields the operating system from the electronic details of the interrupt lines. Some controllers are able to *queue* interrupts, such that none of them gets lost (up to a given hardware limit, of course). Some allow various ways of configuring *priorities* on the different interrupts.

  The 8259 (http://www.cast-inc.com/cores/c8259a/c8259a-x.pdf) *Programmable Interrupt Controller* (PIC) is still a very common chip for this job on PC architectures, despite its age of more than 25 years. PC builders usually use two PICs, since each one can cope with only eight interrupts. But using more than one *has* to happen in a *daisy chain*, i.e., the interrupt output pin of the first PIC is connected to an input pin of the second one; this introduces delays in the interrupt servicing. As another disadvantage, PICs were not designed to be used in multiprocessor systems.

  Higher-quality and/or SMP motherboards use the *Advanced Programmable Interrupt Controller* (APIC). This is not just a single chip, but a small hardware system that manages interrupts:

  - Each CPU must have a "local APIC" with which it gets interrupts from the APIC system.

- The peripheral hardware connects its interrupt line to the *I/O APIC*. (There can be eight of them.) An I/O APIC then sends a signal to the local APIC of the CPU for which the interrupt is meant.

The APIC architecture is better than the PIC, because (i) it can have many more interrupts lines, hence eliminating the need to share interrupts, (ii) it knows programmable interrupt priorities, (iii) it is faster to program (only one machine instruction to the local APIC'c Task Priority Register (which is *on* the CPU!), instead of two to the PIC, which in addition is not on the CPU) and (iv) it allows to work with *level-triggered interrupts* instead of with *edge-triggered interrupts*. The PCI bus uses active low, level-triggered interrupts, so can work fine together with APIC.

The PowerPC platforms have another interrupt hardware standard, the *OpenPIC (http://www.itis.mn.it/inform/materiali/evarchi/cyrix.dir/opnparc.htm)*, which also guarantees a high hardware quality. OpenPIC also works with x86 architectures.

3.3. Interrupt software

From the software side, an interrupt-driven system must typically take into account one or more of the following software issues:

- *Interrupt Service Routine* (ISR), often called *interrupt handler* tout court. This software routine is called when an interrupt occurs on the interrupt line for which the ISR has been *registered* in the interrupt vector. Typically, this registration takes place through a system call to the operating system, but it can also be done directly in a machine instruction, by a sufficiently privileged program. The registration puts the address of the function to be called by the interrupt, in the address field provided in the interrupt vector at the index of the corresponding interrupt number.

  The operating system does not (or rather, cannot) intervene in the launching of the ISR, because everything is done by the CPU. The context of the currently running task is saved on the stack of that task: its address is in one of the CPU registers, and it is the only stack that the CPU has immediate and automatic access to. This fact has influence on the software configuration of the system: each task must get enough stack space to cope with ISR overhead. So, the worst-case amount of extra stack space to be foreseen in a task's memory budget can grow large, especially for systems in which interrupts can be nested. More and more operating systems, however, provide a separate "context" for interrupt servicing, shared by *all* ISRs; examples are Linux and VxWorks.

  An ISR should be as short as possible, because it runs with interrupts disabled, which prevents other interrupts from being serviced, and, hence, other tasks from proceeding. The ISR should service the peripheral device it was triggered by, and then return. This servicing typically consists of reading or writing some registers on the device, and buffer them in a place where some other task can process them further, outside of the ISR and hence with interrupts enabled again. This further processing is the goal of the DSR (Deferred Service Routine), Section 3.4. Getting the data from the ISR to the DSR should be done in a *non-blocking* way; FIFOs (Section 5.2) or circular buffers (Section 5.4) are often used for this purpose.

- *Trap handler/service request.* A synchronous interrupt is sometimes also called a *trap (http://www.osdata.com/topic/language/asm/trapgen.htm)* or a *software interrupt*. The software interrupts are "called" by the processor itself, such as in the case of register overflow, page address errors, etc. They work like a hardware interrupts (saving state, switching to protected mode, jumping to handler), but they run with the hardware interrupts *enabled*.

  These software interrupts are very important, because they are the only means with which user space tasks can execute "protected operations," or "privileged instructions." These privileged instructions can only be executed when the processor is in its *protected mode* (also called *privileged mode*). Privileged instructions are operations such as: to address physical IO directly; to work with the memory management infrastructure such as the page lookup table; to disable and enable interrupts; or to halt the machine. Privileged instructions are available to user space tasks via *system calls*, that are in fact handlers of software interrupts: a system call puts some data in registers or on the stack, and then executes a software interrupt, which makes the processor switch to protected mode and run the interrupt handler. That handler can use the register or stack data for task specific execution. Recall that the handler of a software interrupt runs in the context of the task that executes the system call, so it can read the data that the task has put on the stack. But now the execution takes place in the protected mode of the processor.

  A system call is just one example of a software interrupt, or trap. An interrupt service routine of a trap is often called a *trap handler* . Still another name for a software interrupt is a *service request (SRQ)*. Each type of CPU has part of its interrupt vector reserved for these trap handlers. Operating systems typically have a default trap handler installed, which they attach to all possible software interrupts in your system. Usually, you can replace any of these by your own. For example, RTAI has a `rt_set_rtai_trap_handler()` for this purpose. The OS also reserves a number of traps as *system signals*. For example, RTAI reserves 32 signals, most of them correspond to what standard Linux uses.

  Trap handlers are a major tool in *debugging*: compiling your code with the debug option turned on results, among other things, in the introduction in (the compiled version of) your original code of machine instructions that generate a trap after each line in your code. The ISR triggered by that trap can then inform the debug task about which "breakpoint" in your program was reached, thanks to the register information that the trap has filled in (Section 3.3).

  Another major application of the trap functionality, certainly in the context of this document, is their use by RTAI to deliver user space hard real-time functionality (Section 11.5). Linux just uses one single software interrupt, at address *`0x80`*, for its user space system calls, leaving a lot of software interrupts to applications such as RTAI.

- *Interrupt latency.* This is the time between the arrival of the hardware interrupt and the start of the execution of the corresponding ISR. The latency is not a crisp number, but rather a statistical quantity becaused it is influenced by a large number of undeterministic effects (Section 1.3.2). This becomes more and more the case in modern processors with their multiple levels of caches and instruction pipelines, that all might need to be reset before the ISR can start. This latter fact is at the origin of the somewhat counter-intuitive phenomenon that some modern Gigahertz CPUs have longer interrupt latencies than much older digital signal processors.

- *Interrupt enable/disable.* Each processor has atomic operations to enable or disable ("*mask*") the interrupts. Common names for these functions are `sti()` ("set interrupt enable flag", i.e., enable interrupts to come through to interrupt the CPU) and `cli()` ("clear interrupt enable flag", i.e., don't allow interrupts). In the same context, one finds functions like `save_flags()` and `restore_flags()`. These are a bit more fine-grained than `sti()` and `cli()`, in the sense that they save/restore a bit-sequence where each bit corresponds to an hardware interrupt line and indicates whether or not the interrupt on that particular line should be enabled. In other words, it saves the "state" of the interrupt vector. (`restore_flags()` in some cases does an implicit enabling of the interrupts too.) Note that `cli()` disables *all* interrupts on *all* processors in an SMP system. That is a costly approach to use, particularly so in an RTOS.
- *Interrupt priorities.* Some systems offer, as a *hardware feature*, *(static) priorities* to interrupts. That means that the OS blocks a new interrupt if an ISR of an interrupt with a higher priority is still running. (Or rather, as long has it has not enabled the interrupts again.) Similarly, the ISR of a lower-priority interrupt is pre-empted when a higher-priority interrupt comes in. Hence, ISRs must be *re-entrant.* And, if the processor allows interrupt priorities, most opportunities/problems that are known in task scheduling (see Section 2.4) show up in the interrupt handling too!
- *Prioritized interrupt disable.* Prioritized enabling/disabling of the interrupts is a *software feature* (that must have hardware support, of course) that allows the programmer to disable interrupts below a specified *priority level*. Microsoft NT is an example of an OS kernel that extensively uses this feature.
- *Interrupt nesting.* If the processor and/or operating system allow interrupt nesting, then an ISR servicing one interrupt can itself be pre-empted by another interrupt (which could come from the same peripheral device that is now being serviced!). Interrupt nesting increases code complexity, because ISRs must use *re-entrant* code only, i.e., the ISR must be written in such a way that it is robust against being pre-empted at any time.
- *Interrupt sharing.* Many systems allow different peripheral devices to be linked to the same hardware interrupt. The ISR servicing this interrupt must then be able to find out which device generated the interrupt. It does this by (i) checking a status register on each of the devices that share the interrupt, or (ii) calling in turn all ISRs that users have registered with this IRQ.

  Interrupt sharing is implemented in most general purpose operating systems, hence also in the Linux kernel. (See the file `/kernel/softirq.c`.) Linux accepts multiple interrupt handlers on the same interrupt number. The kernel hangs its own ISR on the hardware interrupt, and that kernel ISR invokes one by one all the handler routines of the ISRs that have been registered by the application programs. This means that they will be executed *after* the hardware ISR has finished, but *before* any other tasks, and with interrupts enabled.

  While the Linux kernel does interrupt sharing as mentioned in the previous paragraph, RTLinux and RTAI don't: they allow only one single ISR per IRQ, in order to be as deterministic as possible. (So, be careful when putting interface cards in your computer, because all the ones for which you want to install *real-time* drivers must be connected to different interrupt lines!) The real-time ISR that a user program has registered is directly linked to the hardware interrupt, and hence runs with all interrupts disabled. The other ISRs on that same IRQ are only executed when the non-real-time Linux kernel on top of the RTOS gets the occasion to run, i.e., after *all* real-time activity is done, also non-ISR activity.

3.4. ISR, DSR and ASR

An ISR should be as short as possible, in order to minimize the delay of interrupts to other ISRs, and the scheduling of tasks. In general-purpose operating systems, only the ISR that the OS has attached to each IRQ runs with interrupts disabled, but not the user-registered ISRs. A real real-time operating system, on the other hand, allows only one ISR per IRQ, otherwise, the time determinism of the other ISRs is not guaranteed! This makes the job for real-time programmers a bit easier, because they can design *non-re-entrant* (and hence often simpler and faster) ISRs: the ISR can store local information without the danger that it can be overwritten by another invocation of the *same* ISR code, and it has the guarantee of *atomicity*, (i.e., the ISR will run without being pre-empted. However, when the OS and the hardware allow interrupt priorities, the ISR at one IRQ level *can* be pre-empted by a higher-priority interrupt.

Typically, a hardware ISR just reads or writes the data involved in the communication with the peripheral device or the trap that caused the interrupt, acknowledges the interrupt if the peripheral device requires it, and then, if needed, wakes up another “task” to do any further processing. For example, most drivers for network cards just transfer the raw packet data to or from the card in the ISR, and delegate all *interpretation* of the data to another task. The skeleton of a typical ISR-DSR combination would look like this:

```
dsr_thread()
{
   while (1) {
      wait_for_signal_from_isr();
      process_data_of_ISR (); // including all blocking stuff
   }
}

interrupt_handler( )
{
   reset_hardware();
   do_isr_stuff();
   send_signal_to_wake_up_dsr();
   re_enable_interrupts() // some RTOSs do this automatically
}
```

In the Linux kernel, this latter task used to be a *bottom half*, while the hardware interrupt-driven ISR was called the *top half*. (Note that some operating systems use opposite terminology.) The bottom half concept is more or less abandoned, and replaced by *tasklets* and *softirqs* (see the files `include/linux/interrupt.h` and `kernel/softirq.c`). The reason for abandoning the bottom halves is that Linux has a hard limit of maximum 32 bottom halves functions. Moreover, they run with locks over the *whole* system, which is not very good for multi-processor systems. The softirq was introduced in the 2.3.43 kernel, as a multi-processor-aware version of the bottom half; there are still only 32 of them, so application programmers should stay away from them, and use *tasklets* instead.

(Tasklet are a very appropriate primitive in the context of interrupt servicing, but its usefulness is in no way limited to only this context!)

"Tasklet" is a bit of an unfortunate name, because it has not much to do with schedulable tasks: a tasklet is a *function* that the kernel calls when an ISR has requested a "follow-up" of its interrupt servicing. Outside of the Linux world, this follow-up function is more often called DSR, *Deferred Service Routine*, or (in Microsoft NT), *Deferred Processing Call*. In Linux, an unlimited number of tasklets is allowed, and they have the same behaviour and functionality as the *softirqs*.

In Linux the ISR requests the execution of a tasklet/DSR via the `tasklet_schedule(&tasklet)` command. The tasklet has first to be initialized with a `tasklet_init (&tasklet, tasklet_function, data)`; this call links a tasklet identifier with the function to be executed and a data structure in which the ISR can store information for processing by the tasklet. The tasklet (or DSR, or softirq) runs with interrupts *enabled*, but outside of the context of a particular task, just as the ISR that has requested it. This means that neither the ISR, nor the DSR can use variables that have been defined locally in the scope of the task(s) to which they logically are related. The execution of tasklets is implemented in the `kernel/softirq.c` file, and both tasklets and softirqs are treated as *`softirq`* tasks.

Linux (and many other general purpose operating systems) executes the DSRs in sequence (without mutual pre-emption, that is), at the end of hardware ISRs and *before* the kernel returns to user space. So, at the end of each kernel call, the scheduler checks whether some DSRs are ready to be executed; see the file `kernel/softirq.c` in the Linux source code.

RTAI also has tasklets, and their semantics is more or less like the Linux tasklets. However, RTAI added some extra features (see the files `include/rtai_tasklets.h` and `tasklets/tasklets.c`):

- There is a special class of tasklets, called *timers*. They can be used to let context-independent functions run with specified timings.
- RTAI also allows a user space function to be executed as a tasklet.

The RTAI tasklets are executed by a dedicated task and/or ISR in the RTAI kernel.

An ISR is not allowed to use semaphores or any other potentially *blocking* system calls: an ISR that blocks on a lock held by another task causes big trouble, because all interrupts are disabled when the ISR runs, such that the condition to wake up the other task might never occur. The same holds for RTAI tasklets: a blocking tasklet also blocks the timer ISR or task that executes all tasklets.

Avoiding non-blocking calls is sufficient for maximum determinism in a UP ("uni-processor") system. In a multi-processor system, however, a race condition (see Section 4.2) can occur between the hardware ISR on one processor, and any other task on one of the other processors; e.g., because the ISR and the other task access shared data. (Remember that the ISR cannot use a lock!) The easiest solution is to not only mask the interrupts for one processor, but for all of them. This, however, prevents *all* processors from working. One way around this are *spinlocks* (see Section 4.6.3). The operating system also helps a bit, by guaranteeing that tasklets are *serialized over all processors* in the system; i.e., only one is executed at a time.

Some operating systems have one more level of interrupt sharing: besides the ISR and DSR functions, they offer the possibility to use *Asynchronous Service Routines (ASR)*. (This name is not as standardized as ISR and DSR.) In Microsoft NT, it is called an *Asynchronous Procedure Call*; eCos calls it DSR; Linux doesn't have the concept. ASRs can run after all DSRs have finished, but before normal tasks get a chance to be scheduled. Their goal is to execute that part of the reaction to an interrupt, that needs the thread's context; for example, to make the thread stop some of its activities, including itself.

The eCos operating system executes an ASR with interrupts enabled, with the scheduler disabled, and always in the context of one specific thread. So, the ASR can call all system functions, which is not the case for ISR and DSR, which are not bound to a deterministically defined context.

The RTAI operating system gives the possibility to add to each real-time task a user-defined function that runs in the task's context and with interrupts disabled, *every* time that the task gets scheduled (hence, not just when an interrupt has occurred). This allows, for example, an interrupt servicing to indirectly change some task-specific attributes at each scheduling instant. This user-defined function is called "`signal()`" and is filled in by `rt_task_init()` (XXX ???) in the task data structure. However, it's just a pointer to a function, so it could be filled in or changed on-line.

Chapter 4. IPC: synchronization

The decision about what code to run next is made by the operating system (i.e., its scheduler), or by the hardware interrupts that force the processor to jump to an associated interrupt routine. To the scheduler of the OS, all tasks are just “numbers” in scheduling queues; and interrupts “talk” to their own interrupt service routine only. So, scheduler and interrupts would be sufficient organizational structure in a system where all tasks just live next to each other, without need for cooperation. This, of course, is not sufficient for many applications. For example, an interrupt service routine collects measurements from a peripheral device, this data is processed by a dedicated control task, the results are sent out via another peripheral device to an actuator, and displayed for the user by still another task.

Hence, the need exists for *synchronization* of different tasks (What is the correct sequence and timing to execute the different tasks?), as well as for *data exchange* between them. Synchronization and data exchange are complementary concepts, because the usefulness of exchanged data often depends on the correct synchronization of all tasks involved in the exchange. Both concepts are collectively referred to as *Interprocess communication* (“IPC”).

The role of the operating system in matters of IPC is to offer a sufficiently rich set of IPC-supporting primitives. These should allow the tasks to engage in IPC without having to bother with the details of their implementation and with hardware dependence. This is not a minor achievement of the operating system developers, because making these IPC primitives safe and easy to use requires a lot of care and insight. In any case, the current state-of-the-art in operating systems’ IPC support is such that they still don’t offer much more than just *primitives*. Hence, programmers have to know how to apply these primitives appropriately when building software systems consisting of multiple concurrent tasks; this often remains a difficult because error-prone design and implementation job. Not in the least because no *one-size-fits-all* solution can exist for all application needs.

4.1. IPC terminology

The general *synchronization* and *data exchange* problems involve (at least) two tasks, which we will call the “sender” and the “receiver”. (These tasks are often also called “writer” and “reader”, or “producer” and “consumer”.) For *synchronization*, “sender” and “receiver” want to make sure they are both in (or *not* in) specified parts of their code at the same time. For *data exchange*, “sender” and “receiver” want to make sure they can exchange data efficiently, without having to know too much of each other (“decoupling”, Chapter 14), and according to several different *policies*, such as blocking/non-blocking, or with/without data loss.

Data exchange has a natural direction of flow, and, hence, the terminology “sender” and “receiver” is appropriate. Synchronization is often without natural order or direction of flow, and, hence, the terminology “sender” and “receiver” is less appropriate in this context, and “(IPC) client” might be a more appropriate because symmetric terminology. Anyway, the exact terminology doesn’t matter too much. Unless we want to be more specific, we will use the generic system calls `send()` and `receive()` to indicate the IPC primitives used by sender and receiver, respectively.

4.1.1. Blocking/Non-blocking

IPC primitives can have different effects on *task scheduling*:

- *Blocking.* When executing the `send()` part of the IPC, the sender task is blocked (i.e., non-available for scheduling) until the receiver has accepted the IPC in a `receive()` call. And similarly the other way around. If both the sender and the receiver block until *both of them* are in their `send()` and `receive()` commands, the IPC is called *synchronous*. (Other names are: *rendez-vous*, or *handshake*.) Synchronous IPC is the easiest to design with, and is very similar to building hardware systems.
- *Non-blocking (asynchronous).* Sender and receiver are not blocked in their IPC commands. This means that there is incomplete synchronization: the sender doesn't know when the receiver will get its message, and the receiver cannot be sure the sender is still in the same state as when it sent the message.
- *Blocking with time out.* The tasks wait in their IPC commands for at most a specified maximum amount of time.
- *Conditional blocking.* The tasks block in their IPC commands only if a certain condition is fulfilled.

Of course, blocking primitives should be used with care in real-time sections of a software system.

4.1.2. Coupling

IPC primitives can use different degrees of *coupling*:

- *Named connection*: sender and receiver know about each other, and can *call each other by name*. That means that the sender fills in the unique identifier of the receiver in its `send()` command, and vice versa. This can set up a connection between both tasks, in a way very similar to the telephone system, where one has to dial the number of the person one wants to talk to.

  The connection can be *one-to-one*, or *one-to-many* (i.e., the single sender sends to more than one receiver, such as for broadcasting to a set of named correspondents), or *many-to-one* (for example, many tasks send logging commands to an activity logging task), or *many-to-many* (for example, video conferencing).

- *Broadcast*: the sender sends its message to all "listeners" (without explicitly calling them by name) on the (sub-branch of the) *network* to which it is connected. The listeners receive the message if they want, without the sender knowing exactly which tasks have really used its message.
- *Blackboard*: while a broadcast is a message on a network-like medium (i.e., the message is not stored in the network for later use), a blackboard IPC *stores* the messages from different senders. So, receivers can look at them at any later time.
- *Object request broker* (ORB): the previous types of IPC all imply a rather high level of *coupling* between sender and receiver, in the sense that they have to know explicitly the identity of their communication partner, of the network branch, or of the blackboard. The current trend towards more *distributed* and *dynamically reconfigurable* computer systems calls for more *loosely-coupled* forms of IPC. The ORB concept has been developed to cover these needs: a sender *component* registers its

interface with the ORB; interested receivers can ask the broker to forward their requests to an appropriate sender (“server”) component, without the need to know its identity, nor its address.

4.1.3. Buffering

IPC primitives can use different degrees of *buffering*, ranging from the case where the operating system stores and delivers all messages, to the case where the message is lost if the receiver is not ready to receive it.

Not all of the above-mentioned forms of IPC are equally appropriate for *real-time* use, because some imply too much and/or too indeterministic overhead for communication and resource allocation.

4.2. Race conditions and critical sections

Often, two or more tasks need access to the same data or device, for writing and/or reading. The origin of most problems with *resource sharing* (or *resource allocation*) in multi-tasking and multi-processor systems is the fact that operations on resources can usually not be performed *atomically*, i.e., as if they were executed as one single, non-interruptable instruction that takes zero time. Indeed, a task that interfaces with a resource can at any instant be pre-empted, and hence, when it gets re-scheduled again, it cannot just take for granted that the data it uses now is in the same state (or at least, a state that is consistent with the state) before the pre-emption. Consider the following situation:

```
data number_1;
data number_2;

task A
{   data A_number;

    A_number = read(number_1);
    A_number = A_number + 1;
    write(number_2,A_number);
}

task B
{  if ( read(number_1) == read(number_2) )
       do_something();
    else
       do_something_else();
    }
}
```

`task B` takes different actions based on the (non-)equality of *`number_1`* and *`number_2`*. But `task B` can be pre-empted in its `if` statement by `task A`, exactly at the moment that `task B` has already read `number_1`, but not yet `number_2`. This means that it has read `number_1` before the pre-emption, and `number_2` after the pre-emption, which violates the validity of the test.

The `if` statement is one example of a so-called *critical section*: it is critical to the validity of the code that the *access to the data* used in that statement (i.e., `number_1` and `number_2`) be executed *atomically*, i.e., un-interruptable by anything else. (Most) *machine code* instructions of a given processor execute atomically; but instructions in higher-level programming languages are usually translated into a sequence of many machine code instructions, such that atomicity cannot be guaranteed.

There are three generic types of critical sections:

- *Access to the same data from different tasks*, as illustrated by the example above.
- *Access to a service.* For example, allocation of a resource, execution of a "transaction" on a database. The service typically has to process a sequence of queries, and these have to succeed as a whole, or fail as a whole.
- *Access to procedure code.* Application tasks often run exactly the same code (for example the control algorithm in each of the joints of a robot), but on other data, and some parts of that code should be executed by one task at a time only.

Of course, many applications involve combinations of different resource sharing needs.

The problem in all above-mentioned examples of access to shared resources is often called a *race condition*: two or more tasks compete ("race") against each other to get access to the shared resources. Some of these race conditions have been given a special name:

- *Deadlock.* `Task A` has locked a resource and is blocked waiting for a resource that is locked by `task B`, while `task B` is blocked waiting for the resource that is locked by `task A`.
- *Livelock.* This situation is similar to the deadlock, with this difference: both tasks are not blocked but are actively trying to get the resource, in some form of *busy waiting*.
- *Starvation.* In this situation, some tasks never get the chance to allocate the resource they require, because other tasks always get priority.

The four conditions that have to be satisfied in order to (potentially!) give rise to a deadlock are:

1. Locks are only *released voluntarily* by tasks. So, a task that needs two locks might obtain the first lock, but block on the second one, so that it is not able anymore to voluntarily release the first lock.
2. Tasks can only get in a deadlock if they need *more than one lock*, and have to obtain them in a (non-atomic) *sequential* order.
3. The resources guarded by locks can only be *allocated to one single task*. (Or to a finite number of tasks.)
4. Tasks try to obtain locks that other tasks have already obtained, and these tasks form a *circular list*. For example, `task A` is waiting for `task B` to release a lock, `task B` is waiting for `task C` to release a lock, and `task C` is waiting for `task A`.

As soon as *one* of these four conditions is not satisfied, a deadlock can not occur. Moreover, these conditions are *not sufficient* for deadlocks to occur: they just describe the conditions under which it is *possible* to have deadlocks.

The literature contains many examples of deadlock *avoidance* and *prevention* algorithms. Deadlock avoidance makes sure that all four necessary conditions are never satisfied at the same time; deadlock prevention allows the possibility for a deadlock to occur, but makes sure that this possibility is never realized. Both kinds of algorithms, however, often require some form of “global” knowledge about the states of all tasks in the system. Hence, they are too indeterministic for real-time execution, and not suitable for *component-based* design (because the requirement for global knowledge is in contradiction with the *loose coupling* strived for in component systems (see Chapter 14).

There are some guaranteed deadlock avoidance algorithms, that are reasonably simple to implement. For example, a deadlock cannot occur if *all programs* always take locks in the same order. This requires a globally known and ordered lists of locks, and coding discipline from the programmers. Other prevention algorithms use some of the following approaches: only allow each task to hold one resource; pre-allocate resources; force release of a resource before a new request can be made; ordering all tasks and give them priority according to that order.

Race conditions occur on a single processor system because of its multi-tasking and interrupt functionalities. But they show up even more on *multi-processor systems*: even if one CPU is preventing the tasks that it runs from accessing a resource concurrently, a task on another CPU might interfere.

4.3. Signals

Signals are one of the IPC synchronization primitives used for *asynchronous notification*: one task fires a signal, which *can* cause other tasks to start doing thins. The emphasis is on “asynchronous” and on “can”:

- *Asynchronous*: the tasks that react to signals are in a completely arbitrary state, unrelated with the signaling task. Their reaction to the signal also need not be instantaneous, or synchronized, with the signaling task. The task that sends the signal, and the tasks that use the signal, need not share any memory, as in the case of semaphores or mutexes. This makes signals about the only synchronization primitive that is straightforward to scale over a *network*.
- *Can*: the signaling task fires a signal, and continues with its job. Whether or not other tasks do something with its signal is not of its concerns. The operating system takes care of the delivery of the signal, and it nobody wants it, it is just lost.

In most operating systems, signals

- are *not queued*. A task’s signal handler has no means to detect whether it has been signaled more than once.
- *carry no data*.

- *have no deterministic delivery time or order*. A task that gets signaled is not necessarily scheduled immediately.
- *have no deterministic order*. A task that gets signaled multiple times has no way to find out in which temporal order the signals were sent.

So, these are reasons to avoid signals for *synchronization between two running tasks*, [BrinchHansen73]. In other words: *notification* in itself is not sufficient for *synchronization*. Synchronization needs two tasks that do something together, while taking notice of each other, and respecting each other's activities. Later sections of the text present IPC primitives that are better suited for synchronization than signals.

POSIX has standardized signals and their connection to threads. The OS offers a number of pre-defined signals (such as "kill"), and task can ask the operating system to connect a handler (i.e., a function) to a particular signal on its behalf. The handler is "registered", using the system call `sigaction()`. The task also asks the OS to receive or block a specific subset of all available signals; this is its "signal mask". Whenever a signal is received by the operating system, it executes the registered handlers of all tasks that have this signal in their mask. The task can also issue a `sigwait(signal)`, which makes it sleep until the `signal` is received; in this case, the signal handler is *not executed*. Anyway, signals are a bit difficult to work with, as illustrated by this quote from the `signal` man page:

> For `sigwait` to work reliably, the signals being waited for must be blocked in all threads, not only in the calling thread, since otherwise the POSIX semantics for signal delivery do not guarantee that it's the thread doing the `sigwait` that will receive the signal. The best way to achieve this is block those signals before any threads are created, and never unblock them in the program other than by calling `sigwait`.

The masks are also set on a *per-thread* basis, but the signal handlers are shared between all threads in a process. Moreover, the implementation of signals tend to differ between operating systems, and the POSIX standard leaves room for interpretation of its specification. For example, it doesn't say anything about the *order* in which blocked threads must be woken up by signals. So, these are reasons why many developers don't use signals too much.

POSIX has a specification for so-called "real-time signals" too. Real-time signals are queued, they pass a 4-byte data value to their associated signal handler, and they are guaranteed to be delivered in numerical order, i.e., from lowest signal number to highest. For example, RTLinux implements POSIX real-time signals, and offers 32 different signal levels. (See the file `include/rtl_sched.h` in the RTLinux source tree.) And RTAI also offers a 32 bit unsigned integer for events, but in a little different way: the integer is used to allow signalling *multiple* events: each bit in the integer is an event, and a task can ask to be notified when a certain AND or OR combination of these bits becomes valid. (See the file `include/rtai_bits.h` in the RTAI source tree.)

4.4. Exceptions

Exceptions are signals that are sent ("raised") *synchronously*, i.e., by the task that is currently running. (Recall that signals are *asynchronous*, in the sense that a task can receive a signal at any arbitrary moment in its lifetime.) Exceptions are, roughly speaking, a signal from a task to itself. As operating system primitive, an exception is a software interrupt (see Section 3.1) used to handle non-normal cases in the execution of a task: numerical errors; devices that are not reachable or deliver illegal messages; etc. The software interrupt gives rise to the execution of an exception handler, that the task (or the operating system, or another task) registered previously. In high-level programming languages, an exception need not be a software interrupt, but it is a function call to the language's *runtime* support, that will take care of the exception handling.

4.5. Atomic operations

The concept of an *atomic operation* is very important in interprocess communication, because the operating system must guarantee that the taking or releasing a lock is done without interruption. That can only be the case if the *hardware* offers some form of atomic operation on bits or bytes. Atomic operations come in various forms: in the hardware, in the operating system, in a language's run-time, or in an application's support library, but always, the hardware atomic operation is at the bottom of the atomic service. This Section focuses on the *hardware* support that is commonly available.

Most processors offer an atomic machine instruction to *test a bit* (or a byte or a word). In fact, the operation not just *tests* the bit, but also *sets* the bit if that bit has not already been set. Hence, the associated assembly instruction is often called `test_and_set()`, or something similar. Expressed in pseudo-code, the `test_and_set()` would look like this:

```
int test_and_set(int *lock){
   int temp = *lock;
   *lock = 1;
   return temp;
}
```

Another atomic instruction offered by (a fewer number of) processors is `compare_and_swap(address,old,new)`: it compares a value at a given memory address with an "old" value given as parameter, and overwrites it with a "new" value if the compared values are the same; in this case, it returns "true". If the values are not equal, the new value is copied over the old value. Examples of processors with a `compare_and_swap()` are the Alpha, ia32/ia64, SPARC and the M68000/PowerPC. (Look in the Linux source tree for the *`__HAVE_ARCH_CMPXCHG`* macro to find them.)

The `compare_and_swap()` operation is appropriate for the implementation of the synchronization needed in, for example, *swinging pointers* (see Section 4.10): in this case, the parameters of the `compare_and_swap(address,old,new)` are the address of the pointer and its old and new values.

The following pseudo-implementation is simplest to understand the semantics of the `compare_and_swap()`:

```
int compare_and_swap(address, old, new) {
  get_lock();
  if (*address == old) {
    *address == new;
    release_lock();
    return (1);
  } else {
    release_lock();
    return (0);
  };
```

The `compare_and_swap()` can, however, be implemented without locks, using the following pair of atomic instructions: `load_linked()` and `store_conditional()`. Together, they implement an atomic read-modify-write cycle. The idea is that the `load_linked()` instruction marks a memory location as "reserved" (but does not lock it!) and if no processor has tried to change the contents of that memory location when the `store_conditional()` takes place, the store will succeed, otherwise it will fail. If it fails, the calling task must decide what to do next: retry, or do something else.

This pair of instructions can be used to implement `compare_and_swap()` in an obvious way, and without needing a lock:

```
int compare_and_swap(address, old, new) {
  temp = load_linked(address);
  if (old == temp) return store_conditional(address,new);
  else return;
}
```

The test `old == temp` need not take place in a critical section, because both arguments are *local* to this single task.

There are some *important caveats* with the `compare_and_swap()` function:

- It only compares the *values* at a given memory location, but does not detect whether (or how many times) this value has changed! That is: a memory location can be changed twice and have its original value back. To overcome this problem, a more extensive atomic operation is needed, the `double_word_compare_and_swap()`, which also checks a *tag* attached to the pointer, and that increments the tag at each change of the value of the pointer. This operation is not very common in processors!
- It is *not multi-processor safe*: (TODO: why exactly?)

While the hardware support for *locks* is quite satisfactory, there is no support for *transaction rollback*. Transaction rollback means that the software can undo the effects of a sequence of actions, in such a way that the complete sequence takes place as a whole, or else is undone without leaving any trace. Transaction rollback is a quite advanced feature, and not supported by operating systems; it's however a primary component of high-end database servers.

4.6. Semaphore, mutex, spinlock, read/write lock, barrier

Race conditions can occur because the access to a shared resource is not well synchronized between different tasks. One solution is to allow tasks to get a *lock* on the resource. The simplest way to lock is to disable all interrupts and disable the scheduler when the task wants the resource. This is certainly quite effective for the running task, but also quite drastic and far from efficient for the activity of all other tasks. Hence, programmers should not use these methods lightly if they want to maintain real multi-tasking in the system. So, this text focuses on locking mechanisms that do *not* follow this drastic approach. Basically, programmers can choose between two types of locking primitives (see later sections for more details):

1. One based on *busy waiting*. This method has overhead due to wasting CPU cycles in the busy waiting, but it avoids the overhead due to bookkeeping of queues in which tasks have to wait.
2. One based on the concept of a *semaphore*. This method has no overhead of wasting CPU cycles, but it does have the overhead of task queue bookkeeping and context switches.

A generic program that uses locks would look like this:

```
data number_1;
data number_2;
lock lock_AB;

task A
{        data A_number;

         get_lock(lock_AB);
         A_number = read(number_1);
         A_number = A_number + 1;
         write(number_2,A_number);
         release_lock(lock_AB);
}

task B
{        get_lock(lock_AB);
         i = ( read(number_1) == read(number_2) );
         release_lock(lock_AB);
         if ( i )
                 do_something();
         else    do_something_else();
         }
}
```

The `get_lock()` and `release_lock()` function calls do not belong to any specific programming language, library or standard. They have just been invented for the purpose of illustration of the idea. When either `task A` or `task B` reaches its so-called *critical section*, it requests the lock; it gets the lock if the lock is not taken by the other task, and can enter the critical section; otherwise, it waits ("blocks", "sleeps") till the other task releases the lock at the end of its critical section. A blocked task cannot be scheduled for execution, so locks are to be used with care in real-time applications: the application programmer should be sure about the *maximum* amount of time that a task can be delayed because of locks held by other tasks; and this maximum should be less that specified by the timing constraints of the system.

The `get_lock()` should be executed *atomically*, in order to avoid a race condition when both tasks try to get the lock at the same time. (Indeed, the lock is in this case an example of a shared resource, so locking is prone to all race conditions involved in allocation of shared resources.) The atomicity of getting a lock seems to be a vicious circle: one needs a lock to guarantee atomicity of the execution of the function that must give you a lock. Of course, (only) the use of an atomic machine instruction can break this circle. Operating systems implement the `get_lock()` function by means of a atomic `test_and_set()` machine instruction (see Section 4.5) on a variable associated with the lock.

Another effective (but not necessarily efficient!) implementation of a lock is as follows (borrowed from the Linux kernel source code):

```
int flags;

save_flags(flags);    // save the state of the interrupt vector
cli();                // disable interrupts
     // ... critical section ...
restore_flags(flags); // restore the interrupt vector to
                      // its original state
sti();                // enable interrupts
```

(Note that, in various implementations, `restore_flags()` implicitly uses `sti()`.)

The implementation described above is not always efficient because: (i) in SMP systems the `cli()` turns off interrupts on *all* CPUs (see Section 3.1), and (ii) if a `test_and_set()` can do the job, one should use it, because the disabling of the interrupts and the saving of the flags generate a lot of overhead.

The lock concept can easily lead to unpredictable latencies in the scheduling of a task: the task can sleep while waiting for a lock to be released; it doesn't have influence on how many locks other tasks are using, how deep the locks are *nested*, or how well-behaved other tasks use locks. *Both* tasks involved in a synchronization using a lock have (i) to agree about which lock they use to protect their common data (it must be in their common address space!), (ii) to be disciplined enough to release the lock, and (iii) to keep the critical section as short as possible. Hence, the locks-based solution to *access or allocation constraints* is equally *indirect and primitive* as the priority-based solution to *timing constraints*: it doesn't protect the *data* directly, but synchronizes the *code* that accesses the data. As with scheduling priorities, locks give disciplined(!) programmers a means to reach deterministic performance measures. But even discipline is not sufficient to guarantee consistency in large-scale systems, where many developers work more or less independently on different parts.

Locks are inevitable for task *synchronization*, but for some common *data exchange* problems there exist *lock-free* solutions (see Section 4.10). The problem with using locks is that they make an application vulnerable for the *priority inversion* problem (see Section 4.8). Another problem occurs when the CPU on which the task holding the lock is running, suddenly fails, or when that task enters a trap and/or exception (see Section 3.1), because then the lock is not released, or, at best its release is delayed.

4.6.1. Semaphore

The name “semaphore” has its origin in the railroad world, where a it was the (hardware) signal used to (dis)allow trains to access sections of the track: when the semaphore was lowered, a train could proceed and enter the track; when entering, the semaphore was raised, preventing other trains from entering; when the train in the critical section left that section, the semaphore was lowered again.

Edsger Dijkstra introduced the semaphore concept in the context of computing in 1965, [Dijkstra65]. A semaphore is an *integer number* (initialized to a positive value), together with a set of function calls *to count* `up()` and `down()`. POSIX names for `up()` and `down()` are `sem_wait()` and `sem_signal()`. POSIX also introduces the *non-blocking* functions `sem_post()` (set the semaphore) and `sem_trywait()` (same as `sem_wait()` but instead of blocking, the state of the semaphore is given in the function’s return value).

A task that executes a `sem_wait()` blocks if the count is zero or negative. The count is decremented when a task executes a `sem_signal()`; if this makes the semaphore value non-negative again, the semaphore unblocks one of the tasks that were blocking on it.

So, the number of tasks that a semaphore allows to pass without blocking is equal to the positive number with which it is initialized; the number of blocked tasks is indicated by the absolute value of a negative value of the semaphore count.

The semaphore *`S`* must also be created (`sem_init(S,initial_count)`) and deleted (`sem_destroy(S)`) somewhere. The *`initial_count`* is the number of allowed *holders* of the semaphore lock. Usually, that number is equal to 1, and the semaphore is called a *binary semaphore.*. The general case is called a *counting semaphore,*. Most operating systems offer both, because their implementations differ only in the initialization of the semaphore’s count.

From an implementation point of view, the minimum data structure of a semaphore has two fields:

```
struct semaphore {
  int count; // keeps the counter of the semaphore.
  queue Q;   // lists the tasks that are blocked on the semaphore.
}
```

And (non-atomic!) pseudo code for `sem_wait()` and `sem_signal()` (for a *binary* semaphore) basically looks like this (see, for example, `upscheduler/rtai_sched.c` of the RTAI code tree for more detailed code):

```
semaphore S;

sem_wait(S)
{
  if (S.count > 0) then S.count = S.count - 1;
  else block the task in S.Q;
}
```

```
sem_signal(S)
{
  if (S.Q is non-empty) then wakeup a task in S.Q;
  else S.count = S.count + 1;
}
```

So, at each instant in time, a negative `S.count` indicates the fact that at least one task is blocked on the semaphore; the absolute value of `S.count` gives the number of blocked tasks.

The semantics of the semaphore as a lock around a critical section is exactly as in its historical railway inspiration. However, a semaphore can also be used for different *synchronization* goals: if `task A` just wants to *synchronize* with `task B`, (irrespective of the fact whether or not it needs to exclude `task B` from entering a shared piece of code), both tasks can use the `sem_wait()` and `sem_signal()` function calls.

Here is a pseudo code example of two tasks `task A` and `task B` that synchronize their mutual job by means of a semaphore:

```
semaphore S;

task A:                      task B:
main()                       main()
{ ...                        { ...
  do_first_part_of_job();      do_something_else_B();
  sem_signal(S);               sem_wait(S);
  do_something_else_A();       do_second_part_of_job();
  ...                          ...
}                             }
```

Finally, note that a semaphore is a lock for which the normal behaviour of the locking task is to go to sleep. Hence, this involves the overhead of context switching, so don't use semaphores for critical sections that should take only a very short time; in these cases *spinlocks* are a more appropriate choice (Section 4.6.3).

4.6.2. Mutex

A mutex (MUTual EXclusion) is often defined as a synonym for a binary semaphore. However, binary semaphore and mutex have an important semantic distinction: a semaphore can be "signaled" and "waited for" by *any* task, while only the task that has *taken* a mutex is allowed to release it. So, a mutex has an *owner*, as soon as it has been taken. This semantics of a mutex corresponds nicely to its envisaged use as a lock that gives *only one task* access to a critical section, excluding all others. That is, the task entering the critical section *takes* the mutex, and *releases* it when it exits the critical section. When another task tries to take the mutex when the first one still holds it, that other task will *block*. The operating systems unblocks one waiting task as soon as the first task releases the mutex. This mutually exclusive access to a section of the code is often also called *serialization*.

A POSIX mutex, for example, is a (counting) semaphore with *priority inheritance* implied (see Section 4.9). The basic POSIX API for mutexes is:

```
pthread_mutex_t lock;
int pthread_mutex_init( // Initialise mutex object:
  pthread_mutex_t *mutex,
  const pthread_mutexattr_t *mutex_attr
);

// Destroy mutex object.
int pthread_mutex_destroy(pthread_mutex_t *mutex);

// Non blocking mutex lock:
int pthread_mutex_trylock(pthread_mutex_t *mutex);

// Blocking mutex lock:
int pthread_mutex_lock(pthread_mutex_t *mutex);

// Mutex unlock:
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

A *recursive mutex* (or *recursive semaphore*) is a mutex that can be locked repeatedly by the owner. Otherwise the thread that holds a mutex and would try to take the mutex again would lock itself, hence leading to a deadlock. This recursive property is useful for complex mutual exclusion situations, such as in *monitors*, Section 15.2.

The POSIX API requires to indicate explicitly that a mutex should be recursive:

```
  pthread_mutexattr_settype(&mutex, PTHREAD_MUTEX_RECURSIVE);
```

Some operating systems (e.g., VxWorks) use the recursive mutex mode as the default. Some offer a so-called a *fast* mutex: such a mutex is locked and unlocked in the fastest manner possible on the given operating system (i.e., it doesn't perform any error checks). A fast mutex can only be locked one single time by `pthread_mutex_lock()`, and *all* subsequent calls cause the calling thread to block until the mutex is freed; also the thread that holds the mutex is locked, which causes a deadlock. So, be careful with using fast mutexes.

Many programmers tend to think that a semaphore is necessarily a more primitive RTOS function than a mutex. This is not necessarily so, because one can implement a *counting semaphore* with a mutex and a condition variable (Section 4.7):

```
int sem_wait(sem_t *sem)
{
   pthread_mutex_lock(&sem->mutex);
   while (sem->count == 0) pthread_cond_wait(&sem->cond, &sem->mutex);
   sem->count--;
   pthread_mutex_unlock(&sem->mutex);
   return(0);
}
```

4.6.3. Spinlocks

A “spinlock” is the appropriate lock mechanism for multi-processor systems, and for use in all kinds of contexts (kernel call, interrupt service routine, etc.). They are phasing out the use of “hard” exclusion methods such as `cli()` and `sti()`, because: (i) these are too “global”, in the sense that they don’t specify the context in which the lock is needed; (ii) it is usually not necessary to disable interrupts in order to protect two tasks from entering a critical section. However, you can not do all kinds of things when running inside a critical section locked by a spinlock! For example, do nothing that can take a “long” time, or that can sleep. Use semaphores or mutexes for this kind of locks.

The task that wants to get a spinlock tries to get a lock that is shared by all processors. If it doesn’t get the lock, it keeps trying (“*busy waiting* ”) till it succeeds:

```
int spinlock(spinlock_t l){
 while test_and_set(l) {};
};
```

So, it’s clear why you shouldn’t do things that take a long time within a spinlock context: another task could be busy waiting for you all the time! An example of a spinlock in the Linux kernel is the “Big Kernel Lock” (Section 1.2): the BKL is a *recursive* spinlock, i.e., it can be locked multiple times recursively. That means that you (possibly in two separate tasks) can lock it twice in a row, but you also have to release it twice after that.

Spinlocks come in three versions:

1. `spin_lock` and `spin_unlock`: the classical mutual exclusion version, allowing interrupts to occur while in the critical section.
2. `spin_lock_irq` and `spin_unlock_irq`: as above, but with interrupts disabled.
3. `spin_lock_irqsave` and `spin_unlock_irqrestore`: as above, but saving the current state flag of the processor.

All of them work on (the address of) variables of the type *`spinlock_t`*. One should call `spin_lock_init()` before using the lock. The spinlock versions that disable interrupts do *not* disable interrupts on the *other* CPUs than the one the calling task is running on, in order not to bring down the throughput of the whole multi-processor system. An example (Linux specific!) of the usage (not the implementation!) of a spinlock with local interrupt disabling is given here:

```
spinlock_t l = SPIN_LOCK_UNLOCKED;
unsigned long flags
```

```
spin_lock_irqsave(&l, flags);
/* critical section ... */
spin_unlock_irqrestore(&l, flags);
```

So, both the concurrency and the multi-processor issues are dealt with. On a uni-processor system, this should translate into:

```
unsigned long flags;
save_flags(flags);
cli();
/* critical section ... */
restore_flags(flags);
```

Note: the POSIX function `pthread_spin_lock()` has this semantics of disabling interrupts.

Spinlocks are a trade-off between (i) disabling all interrupts on all processors (costly, safe, but what you don't want to do on a multi-processor system or a pre-emptable kernel), and (ii) wasting time in busy waiting (which is the only alternative that remains). So, spinlocks work if the programmer is disciplined enough to use them with care, that is for guaranteed *very* short critical sections. In principle, the latency induced by a spinlock is *not* deterministic, which is in contradiction to its use for real-time. But they offer a good solution in the case that the scheduling and context switching times generated by the use of locks, are larger than the time required to execute the critical section the spinlock is guarding.

There is a reason why atomic *test-and-set* operations are not optimal on multi-processor systems built from typical PC architecture processors: the *test-and-set* performed by one processor can make parts of the caches on the other processors invalid because part of the operation involves *writing* to memory. And this cache invalidating lowers the benefits to be expected from caching. But the following implementation can help a bit:

```
int spinlock(spinlock_type l){
  while test_and_set(l) { // enter wait state if l is 1
    while (l == 1) {}     // stay in wait state until l becomes 0
  };
};
```

The difference with the previous implementation is that the `test_and_set()` requires a read *and* a write operation (which *has* to block memory access for other CPUs), while the test `l == 1` requires only a read, which can be done from cache.

4.6.4. Read/write locks

Often, data has only to be protected against concurrent writing, not concurrent reading. So, many tasks can get a read lock at the same time for the same critical section, but only one single task can get a write lock. Before this task gets the write lock, all read locks have to be released. Read locks are often useful to access complex data structures like linked lists: most tasks only read through the lists to find the element they are interested in; changes to the list are much less common. (See also Section 5.5.)

Linux has a reader/writer spinlock (see below), that is used similarly to the standard spinlock, with the exception of separate reader/writer locking:

```
rwlock_t rwlock = RW_LOCK_UNLOCKED; // initialize

read_lock(&rwlock);
/* critical section (read only) ... */
read_unlock(&rwlock);

write_lock(&rwlock);
/* critical section (read and write) ... */
write_unlock(&_rwlock);
```

Similarly, Linux has a *read/write semaphore*.

4.6.5. Barrier

Sometimes it is necessary to synchronize a lot of threads, i.e., they should wait until *all* of them have reached a certain “barrier.” A typical implementation initializes the barrier with a counter equal to the number of threads, and decrements the counter whenever one of the threads reaches the barrier (and blocks). Each decrement requires synchronization, so the barrier cost scales linearly in the number of threads.

POSIX (1003.1-2001) has a `pthread_barrier_wait()` function, and a *`pthread_barrier_t`* type. RTAI has something similar to a barrier but somewhat more flexible, which it calls “*bits*” (see file `bits/rtai_bits.c` in the RTAI source tree), and what some other operating systems call *flags* or *events*. The `bits` is a 32 bit value, that tasks can share to encode any kind of AND or OR combination of binary flags. It can be used as a barrier for a set of tasks, by initializing the bits corresponding to each of the tasks to “1” and letting each task that reaches the barrier reset its bit to “0”. This is similar to a semaphore (or rather, an array of semaphores), but it is not “counting”.

4.7. Condition variable for synchronization within mutex

Condition variables have been introduced for two reasons (which amount basically to one single reason):

1. It allows to make a task sleep until a certain *application-defined logical criterium* is satisfied.
2. It allows to make a task sleep *within* a critical section. (Unlike a semaphore.) This is in fact the same reason as above, because the critical section is needed to evaluate the application-defined logical criterium atomically.

The solution to this problem is well known, and consists of the *combination* of three things:

1. A *mutex lock* (see Section 4.6.2).
2. A *boolean expression*, which represents the above-mentioned logical criterium.

3. A *signal* (see Section 4.3), that other tasks can fire to wake up the task blocked in the condition variable, so that it can re-check its boolean expression.

The lock allows to check the boolean expression “atomically” in a critical section, and to wait for the signal within that critical section. It’s the operating system’s responsibility to release the mutex behind the back of the task, when it goes to sleep in the wait, and to take it again when the task is woken up by the signal.

There exists a POSIX standard for condition variables. Here are some of the major prototypes for the `pthread_cond_wait()` system call, used to make a task wait for its wake-up signal:

```
#include <pthread.h>

   // Initialise condition attribute data structure:
int pthread_condattr_init(pthread_condattr_t *attr);

  // Destroy condition attribute data structure:
int pthread_condattr_destroy(pthread_condattr_t *attr);

   // Initialise conditional variable:
int pthread_cond_init(
  pthread_cond_t *cond,
  const pthread_condattr_t *cond_attr
);

   // Destroy conditional variable:
int pthread_cond_destroy(pthread_cond_t *cond);

   // Wait for condition variable to be signaled:
int pthread_cond_wait(
  pthread_cond_t *cond,
  pthread_mutex_t *mutex
);

   // Wait for condition variable to be signaled or timed-out:
int pthread_cond_timedwait(
  pthread_cond_t *cond,
  pthread_mutex_t *mutex,
  const struct timespec *abstime
);

   // Restart one specific waiting thread:
int pthread_cond_signal(pthread_cond_t *cond);

   // Restart all waiting threads:
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Others system calls that take the same arguments are: `pthread_cond_init()` (initialize the data structure with which a condition variable is built), `pthread_cond_signal()` (signal the fact that a condition variable has changed state), `pthread_cond_broadcast()` (signals the state change to *all* tasks that are waiting for the signal, and wakes them all), `pthread_cond_timedwait()` (wait for the signal, or for a timer to expire, whichever comes first).

The `sem_wait()` of Section 4.6.2 shows a typical application of a condition variable. We repeat the code here for convenience:

```
int sem_wait(sem_t *sem)
{
   pthread_mutex_lock(&sem->mutex);
   while (sem->count == 0) pthread_cond_wait(&sem->cond, &sem->mutex);
   sem->count--;
   pthread_mutex_unlock(&sem->mutex);
   return(0);
}
```

The semaphore has a mutex *`sem->mutex`*, a condition signal *`sem->cond`*, and its particular boolean expression, namely its *`count`* being zero or not. The checking of this condition, as well as the possible decrement of the *`count`*, must be done in a critical section, in order to synchronize access to the semaphore with other tasks. The `pthread_cond_wait()` function makes the calling task block on the condition variable if the boolean expression evaluates to false. The operating system releases the mutex when the task must block, so that other tasks can use the semaphore. When the condition is signaled (this is done by the complementary function `sem_signal()`, which is not given here but that executes a `pthread_cond_broadcast()`), the calling task is woken up and its mutex is activated (all in one atomic operation) such that the woken-up task can safely access the critical section, i.e., check its boolean expression again. The above-mentioned atomicity is guaranteed by the operating system, which itself uses some more internal locks in its implementation of the `pthread_cond_wait()` call.

It is essential that tasks that wake up from waiting on a condition variable, *re-check* the boolean expression for which they were waiting, because nothing guarantees that it is still true at the time of waking up. Indeed, a task can be scheduled a long time after it was signaled. So, it should also be prepared to wait again. This leads to the almost inevitable `while` loop around a `pthread_cond_wait()`.

The `pthread_cond_broadcast()` should be the default way to signal the condition variable, and not `pthread_cond_signal()`. The latter is only an *optimization* in the case that one knows for sure that only one waiter must be woken up. However, this optimization violates the *loose coupling* principle of good software design (Chapter 14): if the application is changed somewhat, the “optimization” of before could well become a bottleneck, and solving the situation involves looking for the `pthread_cond_signal()` calls that can be spread over various files in the application.

However, blindly using `pthread_cond_broadcast()` can also have a negative effect, called the “*thundering herd*” problem: `pthread_cond_broadcast()` can wake up a large number of tasks, and in the case that only one task is needed to process the broadcast, all other woken-up tasks will immediately go to sleep again. That means the scheduler is hidden under a “herd” of unnecessary wake-up and sleep calls. So, Linux and other operating systems introduced policies that programmers can use to give some tasks the priority in wake-ups.

Both semaphores/mutexes and condition variables can be used for *synchronization* between tasks. However, they have some basic differences:

1. Signaling a semaphore has *always* an effect on the semaphore's internal count. Signaling a condition variable can sometimes have no effect at all, i.e., when no task is waiting for it.
2. A condition variable can be used to check an *arbitrary complex* boolean expression.
3. According to the POSIX rationale, a condition variable can be used to make a task wait *indefinitely long*, but spinlocks, semaphores and mutexes are meant for shorter waiting periods. The reason is that `pthread_mutex_lock()` is not a *cancelling point*, while the `pthread_cond_wait()` is.
4. A condition variable is nothing more than a notification to a task that the condition it was waiting for *might* have changed. And the woken-up task *should* check that condition again before proceeding. This check-on-wake-up policy is not part of the semaphore primitive.

4.8. Priority inversion

Priority scheduling and locks are, in fact, contradictory OS primitives: priority scheduling wants to run the highest priority job first, while a mutex excludes *every* other job (so, also the highest priority job) from running in a critical section that is already entered by another job. And these contradictory goals lead to tricky trade-offs. For example, everybody coding multi-tasking systems using priority-based task scheduling and locking primitives should know about the "priority inversion" danger: in some situations, the use of a lock prevents a task to proceed because it has to wait for a lower-priority task. The reason is that a low-priority task (i) is in a critical section for which it holds the lock that blocks the high-priority task, and (ii) it is itself pre-empted by a medium-priority task that has nothing to do with the critical section in which the high- and low-priority tasks are involved. Hence, the name "priority inversion": a medium-priority job runs while a high-priority task is ready to proceed. The simplest case is depicted in Figure 4-1. In that Figure, `task H` is the high-priority task, `task M` the medium-priority task, and `task L` the low-priority task. At time instant *`T1`*, `task L` enters the critical section it shares with `task H`. At time *`T2`*, `task H` blocks on the lock issued by `task L`. (Recall that it cannot pre-empt `task L` because that task has the lock on their common critical section.) At time *`T3`*, `task M` pre-empts the lower-priority task `task L`, and *at the same time* also the higher-priority `task H`. At time *`T4`*, `task M` stops, and `task L` gets the chance again to finish the critical section code at time *`T5`* when, at last, `task H` can run.

Figure 4-1. Priority inversion.

The best-known practical case of a priority inversion problem occurred during the Mars Pathfinder mission in 1997. (More information about this story can be found at http://www.kohala.com/start/papers.others/pathfinder.html or http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html.)

4.9. Priority inheritance and priority ceiling

Operating system programmers have tried to "solve" (not prevent) the priority inversion problem, in two different ways:

- *Priority inheritance.* A low-priority task that holds the lock requested by a high-priority task temporarily "inherits" the priority of that high-priority task, *from the moment the high-priority task does the request*. That way, the low-priority task will not be pre-empted by medium-level priority tasks, and will be able to finish its critical section without holding up the high-priority task any longer than needed. When it releases the lock, its priority drops to its original level, while the high-priority task will now get the lock. The maximum predictable delay is the length of the critical section of the low-priority task.

  Priority inheritance generates *run-time* overhead, because the scheduler has to inspect the priorities of all tasks that access a lock.

- *Priority ceiling.* Every lock gets a priority level corresponding to the priority of the highest-priority task that *can* use the lock. This level is called the *ceiling priority*. Note that it is the *lock* that gets a priority, which it gives to every task that tries the lock. So, when the low-priority task enters the critical section, it *immediately* gets the ceiling priority from the lock, such that it will not be pre-empted by any medium-level priority task. Therefore, another name of the priority ceiling protocol is *instant inheritance*.

  Priority ceiling generates *compile-time* overhead, because it can already at that moment check the priorities of all tasks that will request a lock.

  Priority ceiling has the pleasant property that it simplifies implementation and has small run-time overhead (only the change in priority for the task entering a critical section): the lock *never has to be tested* for being free or not, because any task that tries the lock runs at the highest priority to enter the critical section: any other task that could test the lock would run at the same ceiling priority, and hence would not have been interrupted in its critical section by the task that currently tests the lock. Indeed, both tasks live in the same priority level and are scheduled with a *`SCHED_FIFO`* policy. Instant inheritance also offers a solution to the "deadly embrace" (see Section 4.2) occurring when two tasks lock *nested* critical sections in opposite order: the first task to enter the outermost lock will have the appropriate priority to finish the complete set of nested critical sections.

  A possible problem with priority ceiling is that it makes more processes run at higher priorities, for longer times than necessary. Indeed, the priorities of tasks are changed, *irrespective* of the fact whether another task will try to request the lock or not. This reduces the discriminating effects of using priorities is the first place, *and* it gives rise to "hidden" priority inversion: while task *`L`* gets its priority raised to the ceiling priority because it is involved in a lock with another task *`V`* that has a very high priority, a third task *`H`* not involved in the lock could get pre-empted by *`L`* although its priority is higher and *`V`* is dormant most of the time.

Priority ceiling and inheritance look great at first sight, and they are part of some OS standards: priority ceiling is in the POSIX standard (*`POSIX_PRIO_PROTECT`*), the Real-Time Specification for Java (RTSJ), OSEK, and the Ada 95 real-time specifications. Priority inheritance is also part of standards such as POSIX (*`POSIX_PRIO_INHERIT`*), and the RTSJ. But priority ceiling and inheritance can still not *guarantee* that no inversion or indeterministic delays will occur, [Yodaiken2002], [Locke2002]. Morever, the priority inheritance “feature” gives rise to code that is more complex to understand and certainly to predict. Also, determining *a priori* the ceiling priority for a lock is not an easy matter (the compiler must have access to *all* code that can possibly use a lock!), and can cause portability and extendability headaches.

Priority inversion is always a result of a *bad design*, so it’s much better to *prevent* race conditions instead of “solving” them. However, contrary to the deadlock prevention algorithm (Section 4.2), no similarly simple and guaranteed algorithm for priority inversion is known. So, all an operating system could do to help the programmers is signalling when priority inversion takes place, such that they can improve their design.

Most RTOSes don’t apply priority inversion solutions for every case of sender-receiver synchronization. For example Neutrino (from QNX) uses separate synchronization mechanisms for critical sections (semaphores) and sender-receiver (which is synchronous IPC in QNX). It solves priority inversion only so long as applications use a many-to-one IPC model. As soon as an application uses many-to-many IPC (via a POSIX queue) there is no more prevention of priority inversion. Many-to-many is inherently difficult because the kernel has no way to know which receiver might be ready next, so all it could do would be to raise the priority of all potential listeners (and the processes upon which they are waiting). And this would often result in a logjam as every process was raised to the same priority, invalidating exactly the major reason why priorities were introduced in the first place.

4.10. Lock-free synchronization for data exchange

Some synchronization can also be done *without* locks, and hence this is much more efficient and guaranteed to be deadlock-free, [Herlihy91], [Herlihy93]. Lock-free synchronization uses the `compare_and_swap(address,old,new)` (see Section 4.5), or similar constructs. This functionality is applicable to the manipulation of *pointers*, e.g., to interchange two buffers in one atomic operation, or to do linked list, queue or stack operations.

The following code fragment shows the basic form of this *pointer swinging*:

```
ptr = ...
do {
  old = ptr;
  new = new_value_for_pointer;
while ( !compare_and_swap(ptr,old,new) );
```

If the `compare_and_swap()` returns “false”, the swinging of the pointers should not be done, because some other task has done something with the pointer in the meantime.

Recall the possible problem with `compare_and_swap()`: it only compares the *values* of the addresses, not whether this value has been changed! This means that a double change of the pointer (back to its original value) will not be detected. This occurs quite frequently, i.e., any time when memory space is re-used, e.g., in a stack or a linked list.

Another problem of using `compare_and_swap` for lock-free synchronization is that it is not always the most efficient method available, because it involves the *copying* of a complete data structure before that data structure can be updated without a lock.

Chapter 5. IPC: Data exchange

The previous Chapter looked at the *synchronization* aspect of IPC; this Chapter deals with the mechanisms and policies of *data exchange*. The emphasis is on data exchange for real-time systems.

The *mechanism* of all data exchange IPC is quite similar: the operating system has some memory space reserved for the data that has to be exchanged, and uses some sychronization IPC primitives for reading or writing to that memory space. There is some "object" responsible for the memory and the locks; we call this object the *mediator*, (Section 14.3) or the *channel*. The mediator is really the heart of the data exchange: the IPC clients make function calls on it, but it's the mediator that takes care of memory allocation, buffering, locking, signalling, etc. Although a mediator is more of an *object-oriented design* concept, it is there already in most of the old C code of operating systems. The bad news is that most operating system designers didn't realize that, and hence, they didn't reuse the mediator code when implementing the myriads of IPC forms they developed. . .

It's especially in their *policy* (i.e., the choice of *how* the low-level mechanism is being used) that the different forms of data exchange differ from each other. Below is a non-exhaustive list of policies for data exchange. Almost every possible combination of options is feasible, so it should come as no surprise that operating systems tend to have data exchange primitives with not quite the same API. . .

- *(No) Data loss.* Whether or not everything that the "sender" sends to the "receiver" (or rather, to the IPC mediator object) will indeed be received by the "receiver".
- *(Non)Blocking.* (Also called *(a)synchronous*.) The "sender" and/or "receiver" block until the exchange is finished.
- *One-to-many/many-to-one/many-to-many/one-to-one.* There is one single "sender" and multiple "receivers". Or any variation on this theme.
- *Named/anonymous.* The "sender" must explicitly give the identification of the "receivers", or it must only name the mediator.
- *(Non)Buffered.* The "sender" sends some data to the "receiver", but only *indirectly*: the data is stored in a buffer, from which the "receiver" reads at its own leisure.
- *(Non)Prioritized.* A message can get a priority, and the highest priority message gets delivered first. Similarly, the tasks that wait for a data exchange (hence, which are blocked by the lock of the mediator), can be woken up according to their static priority.

There does exist some standardization in data exchange IPC: POSIX has its standard 1003.1b, in which it specifies message queues, with 32 priorities and priority-based task queues on their locks.

5.1. Shared memory

Two (or more) tasks can exchange information by reading and writing the same area in memory. The main advantage is that the data exchange can take place with *zero copying*, because the “buffer” is just one deep. For the rest, any policy can be implemented on top of shared memory. One area where shared memory is very popular is for the data exchange with peripheral hardware, if possible under *Direct Memory Access (DMA)*.

Avalaible RAM memory is the only limit to the number of independent shared memory IPC “channels”. The shared memory must be reserved from the operating system, and locked into RAM. (See Section 6.1.) If the tasks involved in the shared memory IPC want to know how “fresh” the data in the shared segment is, they have to implement their own handshake protocol themselves, because the operating system gives no indication as to whether data from shared memory has already been accessed or not. One common approach is to put a counter in the shared memory data structure, that indicates how many writes have already taken place. This counter could be a time stamp, which is, in addition, particularly useful for an asynchronous monitor task in user space: that task, for example, plots the data from the real-time task at its own rate, and the time stamps provide a way to keep the data plotted on a correct time line.

5.2. FIFOs

Shared memory has the properties of a so-called *block device*: programs can access arbitrary blocks on the device, in any sequence. *Character devices*, on the other hand, can access the data only in a specified linear sequence. A FIFO (*First-in, First-Out*) is such a character device IPC: its mediator policy is *loss-free*, *non-blocking* (unless the FIFO is empty or full), in principle *many-to-many* but in practice often *1-to-1* (i.e., only one sender and one receiver), and *buffered* (i.e., FIFOs put data in a *pipeline*, where the sender adds data on one end, and the reader reads it at the other end).

Some FIFO implementations support blocking for synchronization at the reader’s end, so the reader gets woken up as soon as new data has arrived. FIFOs that implement blocking have a “task queue” data structure in which blocked tasks can wait.

FIFO’s often also allow *asynchronous* data exchange: a task can register a *FIFO handler* that the operating system executes after data has been put into the FIFO. This is done with exactly the same ISR-DSR principle as for hardware and software interrupts (Section 3.4): the writing of data into the FIFO also fires an event that activates the FIFO handler; this handler will be a *tasklet* (Section 3.4), that, just as in the case of an interrupt, is executed by the operating system before it does its next scheduling.

The boundaries between successive data in the FIFO need not necessarily be sharp, because different blocks might have different sizes. However, in that case, one speaks more often about *mailboxes*, see Section 5.3.

The *mediator* implementing the FIFO uses a lock for mutual exclusion during read or write, and in order to keep the FIFO’s task queue data structures consistent. However, if the FIFO runs between a real-time task and a user space task, the locking problem is very much simplified: the real-time task can never be interrupted by the user task (because it runs in kernel space) so no lock is needed at the real-time side.

5.3. Messages and mailboxes

Messages and mailboxes allow the sender to send data in *arbitrary chunks*, only limited by available memory in the buffer. The message contains some “meta-information” about the size and sender of the message; or whatever data that the message protocol prescribes. This meta-information is, in fact, the only practical difference with FIFOs. From an implementation point of view, FIFOs, messages and mailboxes all look very similar, in the sense that there is the *mediator* object that takes care of the buffer, the locks and the queues of waiting tasks. Moreover, many operating systems make no distinction between messages and mailboxes. If they do make a distinction, it is the following:

- *Message.* The sender puts its message in a memory space it has allocated itself, and then sends the *address* of that memory to the OS, together with an identification of the receiver. The receiver asks the OS whether there are messages for it, and decides to read them or not. Reading a message is done in the same place as where it was written. If desired, a counter on the message data allows for *1-to-many* IPC.

  Be careful with this oversimplified description: passing the address of a data chunk is error-prone (multi-processor systems; virtual memory; . . .).

- *Mailbox.* The sender notifies the OS that it has a message, and gives the identification of the receiver. The OS then *copies* the message to the mailbox of the receiver; this mailbox is a buffer managed by the *operating system*. The receiver task reads the messages in the order of arrival. (Of course, variations on this policy exist.)

A natural extension to messages or mailboxes is *synchronous message passing* , sometimes also called *Send/Receive/Reply* (because that’s what it is called in QNX): the “sender” sends a message, and waits until the “receiver” has acknowledged the reception of the message. Hence, *two* messages are exchanged in this form of IPC. The SIMPL (http://www.holoweb.net/~simpl/) (*Synchronous Interprocess Messaging Project for Linux*) project offers a free software implementation of this form of message passing.

POSIX has standardized an API for messages (with the semantics of what was called “mailboxes” above, i.e., the message queues are managed by the operating system). Here are the basic data structures and prototypes:

```
struct mq_attr {
    long mq_maxmsg;    // Maximum number of messages in queue
    long mq_msgsize;   // Maximum size of a message (in bytes)
    long mq_flags;     // Blocking/Non-blocking behaviour specifier
                       //   not used in mq_open only relevant
                       //   for mq_getattrs and mq_setattrs
    long mq_curmsgs;   // Number of messages currently in queue
};

   // Create and/or open a message queue:
mqd_t mq_open(
  char *mq_name,
  int oflags,
  mode_t permissions,
  struct mq_attr *mq_attr
);

   // Receive a message:
size_t mq_receive(
  mqd_t mq,
  char *msg_buffer,
  size_t buflen,
  unsigned int *msgprio
);

   // Send a message to a queue
int mq_send(
  mqd_t mq,
  const char *msg,
  size_t msglen,
  unsigned int msgprio
);

   // Close a message queue:
int mq_close(mqd_t mq);

   // Get the attributes of a message queue:
int mq_setattr(mqd_t mq, const struct mq_attr *new_attrs,
                                struct mq_attr *old_attrs);

   // Register a request to be notified whenever a message
   // arrives on an empty queue:
int mq_notify(mqd_t mq, const struct sigevent *notification);

   // Destroy a message queue:
int mq_unlink(char *mq_name);
```

5.4. Circular buffers

A circular (or ring) buffer has most of the properties of shared memory, except that (i) its depth is larger than one (i.e., it can contain more than one of the data structures exchanged in the communication). The buffer is usually implemented as an array of communication data structures, and the positions of sender and receiver are indicated by pointers in this array. When one of these pointers reaches the end of the buffer, it swaps back to the start of the buffer and continues from there. So, data is lost when the sender pointer overtakes the reader pointer; data is read multiple times if the reader pointer overtakes the writer pointer. It’s straightforward to use a lock to avoid these situations. In that case, the lock makes the buffer *blocking*. A lock can also be set on each data item in the buffer, in order to avoid concurrent access of the same data.

Two common options for buffers (especially in real-time applications) are:

- *Locking in memory.* The memory used for the buffer should not be swapped out of the physical RAM.
- *"Buffer Half Full" (High water/Low water) interrupt.* The sender and/or receiver tasks can raise an event if the buffer is more than half full or half empty. This event must wake up the other part of the IPC, such that it can take the appropriate actions to prevent the buffer from overflowing or getting empty.

5.5. Swinging buffers

A swinging buffer (or "flip buffer") is two things:

- *An advanced circular buffer.* Instead of using one single shared memory array, a swinging buffer uses two or more. The sender fills up one of the buffers, while the receiver empties another one. Every time one of the tasks reaches the end of its buffer, it starts operating on a buffer that the other task is not using.
- *A deadlock-free "lock."* Both tasks operate on different data structures, hence no locks are used to access the data. Only when the tasks must decide which buffer to use, they use a lock on the buffer pointers, or the corresponding atomic pointer switching operation (see Section 4.10). In this latter case, the "lock" is atomic in hardware, and hence cannot cause any of the problems generated by software locks.

So, a swinging buffer is *non-blocking* but *loss-prone*, because one task can fill or empty the same buffer of the swinging buffer pair multiple times before the other task is ready to switch buffers.

The swinging buffer approach is also known under the name of *read-copy-update (RCU (http://lse.sourceforge.net/locking/rcu/rcupdate_doc.html))*. It can be used as an alternative for *read-write* locks (Section 4.6.4) for "frequent reads/infrequent writes" applications: the readers follow a pointer, and need no locks, while the (less frequent) writer swaps the pointers after having filled in the new data structure.

5.6. Remote Procedure Calls

The previous flavours of IPC can all be catalogued as "low-level": they are implemented with very basic OS primitives, and are usually shielded from the users within system calls. One of the popular IPC mechanisms at the user level are *Remote Procedure Calls* (RPC). With RPC, a user can invoke the execution of a task on a remote computer, as if that task ran on the processor of the calling task. RPC is implemented on top of messages, with a synchronizing hand-shake protocol. Obviously, RPC is not very real-time, but could be useful for embedded systems.

On the other hand, RPC is the simplest form of what is also called *distributed* or *embedded components*: software objects ("*agents*") that can live on any computer on a network, and that tasks can access transparently. There are three major standards in the area of distributed components:

- *CORBA (http://www.corba.org) (Common Object Request Broker Architecture)*. This is a fully platform and vendor independent initiative.
- *DCOM*, controlled by Microsoft.
- *RMI (Remote Method of Invocation)*, from the Java world.

A real-time extension to CORBA has been specified in 2001. It takes care of the *specification* of the determinism required for real-time applications, but needs to map its functionality onto primitives offered on the host operating system. Of course, the absolute time scales of CORBA real-time are longer than those of a stand-alone computer system.

(TODO: more details about CORBA, and real-time CORBA.)

Chapter 6. Memory management

This Chapter explains what *memory management* means, and how it influences the real-time behaviour of an operating system. Non real-time aspects of memory management (virtual memory, swapping, dirty pages management, etc.) are outside the scope of this document.

6.1. Terminology

All tasks need RAM memory to execute. Not only for placing their data, but also for their code and for IPC with other tasks. A computer system offers a (most often) contiguous space of physical RAM, and the MMU (*Memory Management Unit*) of the hardware, and the *Virtual Memory* software of the operating system, help to give a task the impression that it is the only one that uses the memory. And that that memory is (i) larger than the physically available RAM; (ii) distributed (*transparantly to the task*) over a number of physically non-contiguous memory *pages* of fixed size; and (iii) protected from access by other tasks.

But these general-purpose OS requirements are not those of real-time systems, or of embedded systems on processors without MMU. Their concerns are:

- *Fast and deterministic memory management.* The fastest and most deterministic approach to memory management is no memory management at all. This means that the programmers have all physical RAM available as one contiguous block that they can use as they like. This approach is usually only an option for small embedded systems that run a fixed and small number of tasks. Other RTOSs and EOSs offer at least the basic memory management: memory allocation and deletion through system calls.
- *Page locking.* Demand paging is the common approach in general purpose operating systems to distribute the scarce physical RAM over all tasks: each task gets a number of pages in RAM, and the pages it hasn't accessed recently are "swapped out" to make room for pages of other tasks. This swapping is a non-deterministic thing, because it needs access to disk, and most disk controllers have non-deterministic buffering for optimising the average throughput to or from the disk: when the task needs code or data from one of its pages that is currently swapped out, the page has to be retrieved from disk, and often another page in RAM has first to be swapped out to disk. Hence, the MMU of an RTOS *must* lock the pages of real-time tasks in the physical RAM, in order to avoid the paging overhead. POSIX provides the `mlock()` and `mlockall()` function calls to do this locking.

  Page locking is a *Quality of Service* feature of the operating system: it guarantees that tasks have a specified amount of the memory resource at their disposal. In this respect, it is similar to the QoS extensions of the scheduler (see Section 1.3.3).

- *Dynamic allocation.* A task's memory needs can change during its lifetime, such that it should be able to ask the operating system for more memory. The Linux system call for this purpose is `vmalloc()`. (In kernel space!) A real-time memory manager can only make this dynamic allocation of memory deterministic, if the memory pages can be got from a *pool* of free pages locked in the physical RAM. Anyway, dynamic allocation should be used very carefully in any real-time task, because there is no guarantee that the memory pool has enough free pages left to satisfy all requests. This implies, for example, that IPC approaches with dynamic memory allocation needs (such as unlimited mailboxes and messages, see Section 5.3) are to be avoided.

Nothing prevents an operating system from allocating memory in smaller blocks than one page. However, finer and variable-sized granularity implies more complex memory management, memory *fragmentation*, and hence less determinism.

- *Memory mapping.* Real-time and embedded systems typically have to access peripheral devices. The on-board registers in which these devices place their data have to be *mapped* somewhere into the address space of the corresponding device driver task. The POSIX system call to do this mapping is `mmap()`. Typically, this mapping is a configuration activity, and hence need not be done in real-time.
- *Memory sharing.* One of the most efficient ways for tasks to communicate is through shared memory (see Section 6.2). The operating system has two major responsibilities in this area: (i) (de)allocation of the shared memory, and (ii) synchronizing access to that memory by different tasks. The latter topic is discussed in Chapter 4; the former is illustrated later in this Chapter.
- *RAM disks.* In order to avoid the non-deterministic overhead of accessing hard disks (for real-time systems) or the extra cost, extra space, and reduced robustness of mechanical disk devices (for embedded systems), part of the available RAM can used to *emulate* a hard disk. This means that that memory is organized and accessed as a *file system*, as if it would reside on a hard disk.

  When the RAM disk should be able to preserve data when the power is switched off, the embedded system designer implements it in the form of a *flash disk*. This is memory that can be “burned” many thousand times, rather quickly, with very little power, and from within the system code itself. Reburning (“flashing”) is required either for reprogramming the device, or for temporary storage of “non-volatile” data.

  Having the system code in a file system on flash gives the added bonus that the code need not be loaded into RAM, but may be executed in place. This results in shorter start-up times.
- *Stripped libraries.* RAM is a scarce resource in real-time and embedded systems, such that programmers try to use as little of it as possible. Hence, they often use “stripped down” versions of general utility libraries (C library, math library, GUI library, etc.). μlibc is such a low-footprint version of the C library.

6.2. Shared memory in Linux

(TODO: update state of affairs on shared memory! POSIX API for shared memory; sharing between real-time and user space; shared memory managment through locks and/or monitor; `copy_to_user()`, `copy_from_user()`.)

This Section discusses two complementary ways to allocate shared memory in Linux. There is nothing particularly real-time about *using* shared memory; *allocating* shared memory, however, is more controversial: RTLinux doesn't allow to allocate memory on-line, RTAI does.

The shared-memory pool is a block of physical memory set aside at boot time so that Linux does not use it for processes. To set up the pool, you first determine how much physical memory the system has and how much is to be used for shared memory. Normal Linux processes are required to map physical memory into their private address space to access it. To do this, the Linux processes calls `open()` on the memory device `/dev/mem`. After the file descriptor is opened, the Linux process maps the shared memory into its address space using `mmap()`, which returns a pointer to the shared memory as mapped into the Linux process's address space. Once the shared memory is mapped, it may be accessed by dereferencing the pointer. When the process terminates, you use `munmap()` to unmap the shared memory by passing the pointer and the size of its object. Shared-memory access is easier in the kernel space of the real-time Linux variants, since the real-time code executes in kernel space and thus is not required to map physical addresses to virtual addresses.

6.2.1. Allocation at boot time

In this approach, a block of shared memory can be reserved at *boot time*, to prevent Linux from using it for general purposes. The reservation is done using the *`append=`* parameter in LILO (or something similar for other bootloaders). Here is an example of a `/etc/lilo.conf` file that reserves 1 Megabyte for shared memory out of 16 available Megabytes:

```
image=/boot/zImage
label=rtlinux
root=/dev/hda1
append="mem=15m"
```

Linux will use only the *first* 15 Megabytes, and the last Megabyte can be used for shared memory purposes. The *base address* of the shared memory in the above-mentioned example is:

```
#define BASE_ADDRESS (15 * 0x100000)
```

The real-time and user Linux tasks use different ways to access the memory.

A real-time task runs in kernel space, and hence can directly access the memory with its *physical address*. For example, a data structure of type *`my_data`* at the start of the shared memory is accessed as:

```
my_data *ptr;

ptr = (my_data *) BASE_ADDRESS;
ptr->... = ...
```

A user space tasks must use its *virtual address*. This mapping of physical memory into the virtual address space consists of two steps:

- The user space task must "open" the memory, by using the `open()` system call on the device `/dev/mem`:

  ```
  #include <unistd.h>     // POSIX defined open()
  #include <fcntl.h>      // O_RDWR for read and write access
                          // or O_RDONLY for read-only access, etc.

  int fd;                 // file descriptor for the opened memory

  if ((fd = open("/dev/mem", O_RDWR)) < 0 )) {
         // handle possible error here
  }
  ```

- The `mmap()` system call then does the actual mapping.

  ```
  my_data *ptr;

  ptr = (my_data *) mmap (0, sizeof(my_data),
                          PROT_READ | PROT_WRITE,
                          MAP_SHARED,
                          fd, BASE_ADDRESS);
  ```

  The parameters *PROT_READ* and *PROT_WRITE* are POSIX-defined and indicate read and write access; *MAP_SHARED* indicates that memory can be shared with any other task that maps it too. (See the man pages for more details.)

The shared memory is then accessed via the pointer, as in the kernel space task.

```
my_data *ptr;

ptr->... = ...
```

The task must use `munmap()` to un-map the shared memory used for the *my_data* data structure:

```
my_data *ptr;

munmap(ptr, sizeof(my_data));
```

6.2.2. Allocation in kernel space

The mbuff module implements the `/dev/mbuff` device. This device offers shared memory (allocated in the kernel using the `vmalloc`) in kernel as well as in user space. The shared memory does not need to be reserved at the system startup and its size is not limited by memory fragmentation. It is logically (but not physically) contiguous, and is locked in the physical RAM. When you allocate a block, the kernel first grabs the free pages, then if there is not enough of them, starts freeing more, by reducing buffers, disk cache and finally by swapping out to disk some user data and code. For sure this is not a real-time operation—it may take seconds to get something like 100 MB out of 128 MB RAM machine.

6.2.3. Allocation in module

(TODO: latest kernel options for memory allocation; dmaBuffer module. Use this approach preferably at boot time, otherwise you might not be able to find all the requested memory as a contiguous area in RAM.)

(TODO: `copy_from_user()`.)

Chapter 7. Real-time device drivers

An operating system must interface its peripheral devices to its kernel software as well as to the user application software. This should be done in a modular and systematic way, such that all hardware “looks the same” to software applications. The software that takes care of this hardware-independent interfacing are *device drivers*. For the Linux real-time variants, Comedi (Section 7.4) is a successful and steadily growing project for real-time and non real-time device drivers for digital acquisition cards.

In the UNIX world, device drivers are visible through the `/dev/xyz` “files” (where `xyz` stands for a particular device, such as, for example, `hda` for the first hard disk, `ttyS0` for the first serial line, etc.). The 2.4.X kernels have introduced the `devfs` and driverfs (“driver file system”) approaches, which give more structure to the information about the devices that have actually been loaded. But all these things are for *user space,* and hence not relevant for the real-time Linux variants that operate in kernel space.

The bookkeeping aspects of registering a device driver, with major and minor numbers, as well as guidelines for writing device drivers, are explained in detail in the UNIX literature. For the Linux example, Rubini’s *Linux Device Drivers* book ([Rubini2001]) is the major reference.

7.1. Mechanism and policy

A major feature of a good device driver is that it “provides mechanism, not policy.” This means that it should faithfully mimic all the interfacing capabilities of the device (the “mechanism”), but nothing more. It should *not* try to interpret the exchanged data in any possible user context (the “policy”), because that is the job of that user application program itself. Indeed, once a device driver offers a software interface to the mechanism of the device, an application writer can use this mechanism interface to use the device in *one particular way*. That is, some of the data stuctures offered by the mechanism are interpreted in specific physical units, or some of them are taken together because this composition is relevant for the application. For example, a analog output card can be used to generate voltages that are the inputs for the electronic drivers of the motors of a robot; these voltages can be interpreted as setpoints for the desired velocity of these motors, and six of them are taken together to steer one particular robot with six-degrees of freedom. Some of the other outputs of the same physical device can be used by another application program, for example to generate a sine wave that drives a vibration shaker. Or, the robot control program can use a force sensor that is interfaced through a serial line. The force sensor device driver “talks” to both the application program (i.e., the force control algorithm), and the serial line device driver (for which it is a “user application” itself!). It is obvious that the serial line driver should never implement function calls that are only useful in the force sensor driver context. Nevertheless, that’s exactly what happens in many projects with constrained scope, vision and time. . .

As for the other operating system responsibilities discussed in the previous Chapters, writing device drivers for an RTOS or an EOS is not so much different from writing them for a general-purpose OS. Basically, in an RTOS context, one should make sure that all timing delays in the drivers are both *short* and *deterministic,* and every DSR should be an appropriately prioritized thread or handler that waits on an event to become active.

7.2. Device drivers in UNIX

In the UNIX philosophy, all devices are considered as being “files”, and hence, their device drivers share the following functions: `open()`, `close()`, `read()`, `write()`, `read_config()`, `set_config()`. The function call names are operating system independent, and just for demonstration. However, `open()`, `close()`, `read()` and `write()`, are POSIX compliant. The configuration function calls are, in UNIX often taken together in the `ioctl()` function.

`open()` makes the device accessible for programs, while `close()` ends the accessibility. The device can be opened in different modes, such as, for example, *`O_RDONLY`* (“read-only”) *`O_WRONLY`*, (“write-only”), *`O_RDWR`* (“read and write”), and *`O_NONBLOCK`* (“non-blocking”).

`read()` and `write()` interchange data between the peripheral device and the (kernel or application) software: a known datastructure is copied from one place in memory to another. Of course, the exact contents of that data structure depends on the device and/or on the particular use of this device by the programmer.

`read_config()` reads out the device’s current configuration status, and `set_config()` programs that configuration. Configuration management often has less strict timing constraints than reading and writing. It also has less standardized function calls, because of the larger variety in possible settings of different hardware. Nevertheless, the POSIX standard prescribes the use of the `ioctl()` function call, for all configuration actions that don’t fit cleanly in the classic UNIX stream I/O model of `open()`, `close()`, `read()`, and `write()`:

```
int ioctl(int d, int request, ...)
```

The *`d`* parameter is the “file descriptor” with which the device has been opened; *`request`* is the particular configuration identifier; and `...` are possible arguments that come with the *`request`*.

7.3. Complex device drivers

A simple device driver need nothing more than writing and/or reading of some hardware registers on a peripheral device. Some devices interact with the software through hardware interrupts. Hence, their device drivers must include an ISR, and possibly also a DSR (see Section 3.4). Recall that only a subset of all kernel space functions are available in the run-time context of an ISR. And a real-time device driver is subjected to even more constraints.

Almost all devices can be interfaced in *Programmed Input/Output (PIO)* mode: the processor is responsible for accessing bus addresses allocated to the device, and to read or write data. Some devices also allow shared memory, or even *Direct Memory Access (DMA)*: the device and the memory exchange data amongst each other directly, without needing the processor. DMA is a feature of the bus, not of the operating system; the operating system, however, must support its processes to use the feature, i.e., provide a system call to initialize DMA transfer, and a handler to react to the notification of the device that it has finished its DMA. Anyway, support for shared memory and DMA makes a device driver again a bit more complex.

From the point of view of system developers, it is worthwhile, in the case of complex devices or systems with lots of devices, to standardize the structure and the API for the device drivers as much as possible:

- *API*: devices that offer similar mechanism, should have the same software interface, and their differences should be coped with by parameterizing the interfaces, not by changing the interface for each new device in the family.
- *Structure*: many electronic interfaces have more than one layer of functionality between the hardware and the operating system, and the device driver code should reflect this fact. For example, many different interface cards use the same PCI driver chips, or use the parallel port to connect to the hardware device. Hence, providing “low-level” device drivers for these PCI chips and parallel ports allows for an increased modularity and re-useability of the software. And the mechanism of the low-level drivers is used with different policies in the various higher-level drivers.

7.4. Comedi

David Schleef started the Comedi (http://stm.lbl.gov/comedi/) project to interface lots of different cards for measurement and control purposes. This type of cards are often called *Data Acquisition* cards, or DAQ cards. Schleef designed a structure which is a balance between *modularity* (i.e., it’s fairly easy to integrate a new card because most of the infrastructure part of the driver can be reused) and *complexity* (i.e., the structure doesn’t present so much overhead that new contributors are scared away from writing their new drivers within the Comedi framework).

Comedi works with a standard Linux kernel, but also with its real-time extensions RTAI and RTLinux.

The Comedi project consists of two packages, and three parts: the “comedi” package contains the drivers, and the kcomedilib kernel module for Linux (which is an library to use the drivers in real-time); the “comedilib” package implements the user space access to the device driver functionality.

The cards supported in Comedi have one or more of the following features: *analog input* channels, *analog output* channels, *digital input* channels, and *digital output* channels. The digital channels are conceptually quite simple, and don’t need much configuration: the number of channels, their addresses on the bus, and their direction (input/output).

The analog channels are a bit more complicated. Typically, an analog channel can be programmed to generate or read a voltage between a lower and an upper threshold (e.g., -10V and +10V); the card’s electronics can be programmed to automatically sample a set of channels, in a prescribed order; to buffer sequences of data on the board; or to use DMA to dump the data in an available part of memory, without intervention from the processor.

Many interface cards have extra functionality, besides the analog and digital channels. For example, an EEPROM for configuration and board parameters, calibration inputs, counters and timers, encoders (= quadrature counter on two channels), etc. Therefore, Comedi offers more than just analog and digital data acquisition.

The kernel space structures that Comedi uses have the following hierarchy:

- *channel*: the lowest-level component, that represents the properties of one single data channel (analog in or out; digital in or out). Each channel has parameters for: the voltage range, the reference voltage, the channel polarity (unipolar, bipolar), a conversion factor between voltages and physical units.
- *sub-device*: a set of functionally identical channels that are physically implemented on the same (chip on an) interface card. For example, a set of 16 identical analog outputs. Each sub-device has parameters for: the number of channels, and the type of the channels.
- *device*: a set of sub-devices that are physically implemented on the same interface card; in other words, the interface card itself. For example, the National Instruments 6024E device has a sub-device with 16 analog input channels, another sub-device with two analog output channels, and a third sub-device with eight digital inputs/outputs. Each device has parameters for: the device identification tag from the manufacturer, the identification tag given by the operating system (in order to discriminate between multiple interface cards of the same type), the number of sub-devices, etc.

The basic functionalities offered by Comedi are:

- *instruction*: to synchronously perform one single data acquisition on a specified channel, or to perform a configuration on the channel. “Synchronous” means that the calling process blocks until the data acquisition has finished.
- *scan*: repeated instructions on a number of different channels, with a programmed sequence and timing.
- *command*: start or stop an autonomous (and hence asynchronous) data acquisition (i.e., a number of scans) on a specified set of channels. “Autonomous” means: without interaction from the software, i.e., by means of on-board timers or possibly external triggers.

This command functionality is not offered by all DAQ cards. When using RTAI or RTLinux, the command functionality is emulated through the `comedi_rt_timer` virtual driver. The command functionality is very configurable, with respect to the choice of events with which to signal the progress of the programmed scans: external triggers, end of instruction, etc.

Comedi not only offers the API to access the functionality of the cards, but also to query the capabilities of the installed Comedi devices. That is, a user process can find out on-line what channels are available, and what their physical parameters are (range, direction of input/output, etc.).

Comedi contains more than just procedural function calls: it also offers event-driven functionality. The data acquisition can signal its completion by means of an interrupt or a *callback* function call. Callbacks are also used to signal errors during the data acquisition or when writing to buffers, or at the end of a scan or acquisition that has been launched previously to take place asynchronously (i.e., the card fills up som shared memory buffer autonomously, and only warns the user program after it has finished).

The mechanisms for synchronization and interrupt handling are a bit different when used in a real-time context (i.e., with either RTAI or RTLinux), but both are encapsulated behind the same Comedi calls.

Because multiple devices can all be active at the same time, Comedi provides (non-SMP!) locking primitives to ensure atomic operations on critical sections of the code or data structures.

Finally, Comedi offers the above-mentioned “high-level” interaction, i.e., at the level of user space device drivers, through file operations on entries in the `/dev` directory (for access to the device’s functionality), or interactively from the command line through the “files” in the `/proc` directory (which allow to inspect the status of a Comedi device). This high-level interface resides in the “comedilib” tarball, which is the user space library, with facilities to connect to the kernel space drivers residing in the “comedi” tarball.

7.4.1. Writing a Comedi device driver

(TODO: describe series of steps.)

7.5. Real-time serial line

A real-time device driver for the serial lines is integrated into RTAI. There used to be an independent project, rt_com (http://rt-com.sourceforge.net/), but the developers joined the RTAI bandwagon, and the code was thoroughly rewritten. (Under supervision of the Comedi maintainer, David Schleef.)

The RTAI device driver resides in the “SPdrv” (Serial Port driver) sub-directory of RTAI. It provides very configurable address initialization, interrupt handling, buffering, callbacks, and non-intrusive buffer inspection. It’s a nice purely “mechanism” device driver.

7.6. Real-time parallel port

A real-time device driver for the parallel port is integrated into Comedi. It’s not much different from a user space driver, except for the real-time interrupt handler that can be connected to the interrupt that can be generated on pin 10 of the parallel port. The driver does *not* support *ECP/EPP* parallel ports.

7.7. Real-time networking

The *rtnet* project used to be stand-alone, but is now also integrated into RTAI. It provides a *common programming interface* (real-time and user space) between the RTOS and the device drivers of ethernet cards. Of course, TCP is not supported, due to its inherently non-realtime specifications; UDP is supported.

Although about every possible ethernet card has a Linux driver, these cannot be used unchanged for hard real-time, because their interrupt handling is not real-time safe. Only a couple of the most popular cards are supported, and there is not much interest from the community to port more drivers.

The *CAN bus* is a two-wire bus with a 1Mbits/sec maximum transmission rate, that has become very popular in many industries, such as the automotive. It can be used for real-time, thanks to its CSMA/CD-NDBA bus arbitration protocol. CSMA/CD-NDBA stands for *Carrier Sense Multiple Access with Collision Detect—Non-Destructive Bit Arbitration.* CSMA is also used for ethernet: all clients of the bus sense what is happening on the bus, and stop transmitting when they sense a collision of messages from different clients. The CAN bus adds, in hardware, the NDBA part: this guarantees that the bit sent on the bus is not destroyed in a collision. In CAN the *dominant bit* is the logical “0”, and it overrides the *recessive bit* (logical “1”). So the client that sends a dominant bit will see this dominant bit on the bus, and can continue sending. Each client on the CAN bus has a unique and statically defined identifier of 11 bits wide (29 bits in the extended version of the standard), that corresponds to its *priority*. That means that the client with the most dominant bits early on in its identifier will be the one that survives the NDBA the longest, and hence it is the one that gets the bus first. So, the CAN bus implements “priority-based scheduling” of its clients. Due to the hardware limitations that must guarantee the above-mentioned procedure of surviving dominant bits, a CAN bus has a maximum length of about 100 meters.

The Real-time Transport Protocol (RTP) (http://www.cs.columbia.edu/~hgs/rtp/) (RFC 1889 (ftp://ftp.isi.edu/in-notes/rfc1889.txt)) and the *Real-Time Publish-Subscribe* protocol (drafted by Real-Time Innovations, and to be adopted by the IDA (http://www.ida-group.org)) are “policies” on top of the *normal* ethernet protocol. Hence, despite their names, they will at most be soft real time.

Chapter 8. RTAI: the features

This Chapter introduces the RTAI real-time operating system, as an illustration of the concepts and terminology introduced in the previous Chapters. It describes which features are available in RTAI, and how the API looks like. This Chapter doesn't aim to be a reference or user manual of all RTAI commands; you should look for those manuals you on the RTAI webpage. (http://www.aero.polimi.it/~rtai/documentation/index.html)

8.1. Overview

RTAI consists of five complementary parts:

1. The *HAL (Hardware Abstraction Layer)* provides an interface to the hardware, on top of which both Linux and the hard real-time core can run.
2. The *Linux compatibility layer* provides an interface to the Linux operating system, with which RTAI tasks can be integrated into the Linux task management, without Linux noticing anything.
3. *RTOS core.* This part offers the hard real-time functionality for task scheduling, interrupt processing, and locking. This functionality is not really different from other real-time operating systems.
4. *LX/RT (Linux Real-Time).* The modularity offered by a Hardware Abstraction Layer separated from a core built on top of it, is used in other operating systems too, e.g., eCos. The particular thing about RTAI is the LX/RT component, that makes soft and hard real-time features available to user space tasks in Linux. LX/RT puts a strong emphasis on offering a *symmetric* real-time API: the same real-time functionality should be useable with the same function calls from user space as well as from kernel space. And also the IPC that LX/RT offers between user space and kernel space real-time tasks works with a symmetric API.
5. *Extended functionality packages.* The core is extended with useful extras, such as: several forms of inter-process communication, network and serial line drivers; POSIX interface; interfaces to domain-specific third-party toolboxes such as Labview, Comedi (Section 7.4) and Real-Time Workshop; software watchdogs; etc.

This Chapter explains what features are available in each of these major RTAI parts, as of the 24.1.9 version of RTAI (May 2002). Details about the exact function prototypes can be found in the RTAI reference manual. The following Chapter discusses their *implementation*. The discussion is categorized according to the contents of the previous Chapters of this document. In summary, the feature set of RTAI is quite complete, offering almost all previously presented concepts. RTAI also implements some POSIX parts (Section 1.5): it has POSIX 1003.1c compliant pthreads, mutexes and condition variables, and POSIX 1003.1b compliant pqueues. But POSIX compliance is not high on the priority list of new developments. (A property that RTAI shares with standard Linux development, by the way.)

8.2. Task management and scheduling

Summary: RTAI offers the whole variety of real-time tasks and schedulers. Besides normal tasks that end up in the scheduler queues of the operating system, RTAI offers also non-schedulable units of execution: *tasklets*, *timers*, *ASRs*, and *queue blocks*.

8.2.1. Task management

RTAI has its own specific API, but offers POSIX wrappers for threads. A task is created with the following function:

```
int rt_task_init (
   RT_TASK *task,
   void (*rt_thread)(int),
   int data,
   int stack_size,
   int priority,
   int uses_fpu,
   void(*signal)(void)
);
```

The function's arguments are:

- `task` is a pointer to an `RT_TASK` type structure whose space must be provided by the application. It must be kept during the whole lifetime of the real time task.
- `rt_thread` is the entry point of the task function. The parent task can pass a single integer value `data` to the new task.
- `stack_size` is the size of the stack to be used by the new task.
- `priority` is the priority to be given the task. The highest `priority` is 0, while the lowest is `RT_LOWEST_PRIORITY`.
- `uses_fpu` is a flag. Nonzero value indicates that the task will save the floating point registers at context switches. On a *(multi) uni-processor*, a real-time task does *not* save its floating point context by default. However, when the task is created for a *symmetric multi-processing* system, the floating point context *is* saved, because the task's context must be save against CPU migration anyway.
- `signal` is an "ASR" function (Section 3.4) that is called, within the task environment and with interrupts disabled, when the task becomes the current running task after a context switch.

Here is a typical RTAI code for creating and starting a real-time task, from within an `init_module()`, that periodically runs the function whose code is in the application dependent function `fun()`:

```
#define STACK_SIZE 4096
static RT_TASK mytask;                                          ❶

int init_module(void)
{
  rt_task_init(&mytask, fun, 0, STACK_SIZE, 0, 1, 0);          ❷
  rt_set_runnable_on_cpus(&mytask, ...);                        ❸
  rt_linux_use_fpu(1);                                          ❹
  now = rt_get_time();                                          ❺
  rt_task_make_periodic( \                                      ❻
    &mytask, now + 2000, 100*1000*1000);
  return 0;
}

// function that runs periodically in
// the created real-time task:
static void fun(int t) {                                        ❼
  ...
  while (...) {
    ... // do what has to be done each period
    rt_task_wait_period();                                      ❽
  }
}
```

❶ The task's data structure.

❷ Initialize the task's data structures, giving it, among other things, values for stack space and static priority.

❸ Ask OS to run the task on a *selection* of specified processors.

❹ Ask the OS to save the floating point registers when switching contexts.

❺ Read in the current absolute time.

❻ Ask the OS to run this task periodically. This call also sets the first time instant that the thread wants to become active, and its period. These times are in *nanoseconds*.

❼ The function to be run in the real-time task.

❽ Go to sleep until the schedulers wakes you up when your timer expires.

Using RTAI's wrappers for POSIX pthreads, a typical skeleton for task ("thread") creation and activation looks like this:

```
static pthread_t       thread; // POSIX data structures for task,
static pthread_mutex_t mutex;  // ... mutex,
static pthread_cond_t  cond;   // ... and condition variable,

int init_module(void)
{
  pthread_attr_t attr;         // POSIX data structure for
                               // task properties

  pthread_attr_init  (&attr);        // initializations
  pthread_mutex_init (&mutex, NULL); // ...
  pthread_cond_init  (&cond, NULL);  // ...

  pthread_create (&thread, &attr, fun, 0);                      ❶

  ... // doing stuff until it's time to delete thread

  pthread_mutex_lock (&mutex);
  pthread_cond_signal (&cond);                                  ❷
  pthread_mutex_unlock (&mutex);
  ...
}

// function that runs periodically in
// the created real-time task:
static void fun(int t) {
  #define TIME_OUT 10000000

  ...
  struct sched_param p;
  p.sched_priority = 1;
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);  ❸

  ...
  while (1) {
    time = ...;
    pthread_mutex_lock (&mutex);
    pthread_cond_timedwait (&cond, &mutex, time+TIME_OUT));❹
    pthread_mutex_unlock (&mutex);
    ... // do what has to be done each period
  }
}
```

❶ Here, the task is created, i.e., its data structures and function to execute are initialized.

❷ The created task is signaled; the signal is typically the notification that the thread should stop itself, in a clean way.

❸ The thread itself fills in its scheduling properties. (This could also be done by another task.)

❹ This command makes the task sleep until the specified next wake-up time, or until it receives the signal to clean up. (This signal could have another, task-dependent interpretation too, of course.)

The function `pthread_cond_timedwait()` is used to, both, wait for a time to expire, and for a condition to be signaled (Section 4.7):

```
int pthread_cond_timedwait(
   pthread_cond_t *cond,      // condition variable
   pthread_mutex_t *mutex,    // mutex to protect scope
   struct timespec *abstime);// absolute time to wake up
```

So, the semantics of the pure RTAI and the POSIX implementations are not exactly the same. Another difference between both versions is that a POSIX thread initialization makes the task active immediately, while the task created by a `rt_task_init()` is suspended when created, and must be activated explicitly. (This is achieved by the second argument in `rt_task_make_periodic()`: it specifies the time when the task will be first woken up.)

8.2.2. Tasklets

The data structure to hold the status and the data connected to a tasklet (Section 2.3) is created with the following function:

```
struct rt_tasklet_struct *rt_init_tasklet(void)
```

It is configured with

```
int rt_insert_tasklet(
   struct rt_tasklet_struct *tasklet,  // data structure
   int priority,                       // static priority
   void (*handler)(unsigned long),     // function to execute
   unsigned long data,                 // data to pass to handler
   unsigned long id,                   // user-defined identifier
   int pid)                            // OS process identifier
```

There also exist function calls with which one can set most of the above-mentioned properties separately. RTAI executes the tasklets before it runs its scheduler. And tasklets can set priorities to influence the order in which the operating system executes them. An application can also execute a tasklet explicitly (or rather, wake it up for execution) by a `rt_tasklet_exec(tasklet)` function call. Tasklets do *not* save their floating point registers by default.

8.2.3. Timers

These are nothing else but timed tasklets, so its interface functions have the same semantics as those of tasklets. `rt_init_timer()` is in fact a copy of the `rt_init_tasklet()` function. The major difference lies in the `rt_insert_timer()` function, which inserts the timer tasklet in a list of timers to be processed by a *time manager* task. This function has two more parameters than `rt_insert_tasklet`, which give the tasklet the semantics of a timer:

```
int rt_insert_timer(
   struct rt_tasklet_struct *timer,
   int priority,
   RTIME firing_time,         // fire time
   RTIME period,              // period, if timer must be periodic
   void (*handler)(unsigned long),
   unsigned long data,
   int pid)
```

The *`pid`* parameter is not needed, because the timer tasklet will never be referred to as a "real" task anyway, i.e., as a task that is scheduled by the scheduler. So, some of the fields in the timer data structure (which is equal to the tasklet data structure) are not used. The timer list is ordered according to the desired fire time of the timer tasklets. The time manager always inherits the priority of the highest-priority timer.

8.2.4. ASR

Via the *`signal`* parameter of `rt_task_init()`, the application programmer can register a function that will be executed whenever the task it belongs to will be scheduled, and *before* that task is scheduled. This is the functionality of what is sometimes called an *Asynchronous Service Routine* in other operating systems (Section 3.4). An ASR is different from a tasklet, in the following sense:

- the ASR's function is executed in the context of the task it belongs to, while a tasklet has its own context.
- The ASR is run with interrupts disabled. (This is not always the case for ASRs in other operating systems.)
- The ASR is not a schedulable task itself, i.e., it will never show up in the scheduling queues, just like the timer tasklets.

8.2.5. Queue blocks

(TODO: what is the real use of queue blocks? Seems to be a primitive that somebody happened to have implemented (on QNX) and ported to RTAI without filling a real need?)

Queue blocks are simple structures that contain a pointer to a function and the time at which the function must be executed. The queue blocks are linked into a list and a family of functions are provided to manage the whole thing. The functions are of the type `void (*handler)(void *data, int event)`, and therefore the simple structures also include the arguments data and event. The application may or may not use any of the arguments.

8.2.6. Task scheduling

RTAI provides several complementary scheduling configuration options:

- Depending on the hardware, the following scheduling options are available: uni-processor scheduling (UP), multi-processor scheduling (MUP; the application programmer can assign each task to a specific (set of) processors), and symmetric multi-processor systems (SMP; the scheduler assigns tasks at run-time to any available processor).
- Tasks can configure *periodic scheduling* (scheduled every time a certain time has elapsed) and *one-shot scheduling* (scheduled only once at the requested time).
- RTAI has static priority-based scheduling ("*`SCHED_FIFO`*") as its default hard real-time scheduler, but if offers also Round Robin time-sliced scheduling ("*`SCHED_RR`*"), *Rate Monotonic Scheduling*, and *Earliest Deadline First*. It's the responsibility of the *application programmer* to get the scheduler and timings choices correct. When multiple scheduler schemes are used, RTAI has made the (arbitrary) choice to give EDF tasks a higher priority than tasks scheduled with other policies.

By definition (Section 2.5), only *`SCHED_FIFO`* and *`SCHED_RR`* can be chosen on a per task basis, and with a per task quantum time (only relevant for *`SCHED_RR`*):

```
 rt_set_sched_policy(
   RT_TASK *task,     // pointer to task's data structure
   int policy,        // 0: RT_SCHED_FIFO, 1: RT_SCHED_RR
   int rr_quantum_ns  // RR time slice in nanoseconds, lying between
                      // 0 (= default Linux value) and
                      // 0x0FFFFFFF (= 1/4th of a second)
),
```

(Needing Round Robin scheduling in an application program should be considered as an indication that the program logic is poorly designed. . .) The EDF and RMS schedulers need *global* information about the task timings, so the procedures are a little bit more complex:

- *RMS*: the RMS scheduler is (re)initialized by the function `void rt_spv_RMS(int cpuid)`, to be called *after* the operating system knows the timing information of *all* your tasks. That is, after you have made all of your tasks periodic at the beginning of your application, or after you create a periodic task dynamically, or after changing the period of a task. The *`cpuid`* parameter of the function `rt_spv_RMS()` is only used by the multi uni-processor scheduler.
- *EDF*: this scheduler must know the *start* and *termination* times of all your tasks, so a task must call the function

  ```
  void rt_task_set_resume_end(RTIME resume_time, RTIME end_time);
  ```

  at the end of *every* run of one cycle of the task.

RTAI provides several function calls that influence task scheduling (`ABCscheduler/rtai_sched.c`):

```
void rt_task_yield(void);
   // stops the current task and takes it at the end of the list of
   // ready tasks, with the same priority. The scheduler makes the
   // next ready task of the same priority active.

int rt_task_suspend(RT_TASK *task);
   // suspends execution of the "task". It will not be executed
   // until a call to "rt_task_resume()" or
   // "rt_task_make_periodic()" is made.

int rt_task_resume(RT_TASK *task);
   // resumes execution of the "task" previously suspended by
   // "rt_task_suspend()" or makes a newly created task ready to run.

int rt_task_make_periodic(
  RT_TASK *task,
  RTIME start_time,
  RTIME period);
   // mark the "task" as available for periodic execution, with
   // period "period", when "rt_task_wait_period()" is called.
   // The time of the task's first execution is given by
   // "start_time", an absolute value measured in clock ticks.
```

```
int rt_task_make_periodic_relative_ns(
  RT_TASK *task,
  RTIME start_delay,
  RTIME period);
   // As "rt_task_make_periodic", but with "start_delay" relative
   // to the current time and measured in nanosecs.

void rt_task_wait_period(void);
   // suspends the execution of the currently running task until
   // the next period is reached. The task must have been previously
   // marked for execution with "rt_task_make_periodic()" or
   // "rt_task_make_periodic_relative_ns()".
   // The task is suspended only temporarily, i.e. it simply gives up
   // control until the next time period.

void rt_task_set_resume_end_times(RTIME resume, RTIME end);
int rt_set_resume_time(RT_TASK *task, RTIME new_resume_time);
int rt_set_period(RT_TASK *task, RTIME new_period);

RTIME next_period(void);
   // returns the time when the caller task will run next.

void rt_busy_sleep(int ns);
   // delays the execution of the caller task without giving back
   // the control to the scheduler. This function burns away CPU
   // cycles in a busy wait loop. It may be used for very short
   // synchronization delays only. "nanosecs" is the number of
   // nanoseconds to wait.

void rt_sleep(RTIME delay);
   // suspends execution of the caller task for a time of
   // "delay" internal count units. During this time the CPU is
   // used by other tasks.
   // A higher priority task or interrupt handler can run
   // during the sleep, so the actual time spent in this function
   // may be longer than the specified time.

void rt_sleep_until(RTIME time);
   // similar to "rt_sleep", but the parameter "time" is the
   // absolute time untill when the task is suspended. If the
   // given time is already passed this call has no effect.

int rt_task_wakeup_sleeping(RT_TASK *task);
```

The status of a task can be found with:

```
int rt_get_task_state (RT_TASK *task);
```

The task state is formed by the bitwise OR of one or more of the following flags:

- *READY*: task is ready to run (i.e. unblocked).

- *`SUSPENDED`*: task is suspended.
- *`DELAYED`*: task waits for its next running period or expiration of a timeout.
- *`SEMAPHORE`*: task is blocked on a semaphore.
- *`SEND`*: task sent a message and waits for the receiver task.
- *`RECEIVE`*: task waits for an incoming message.
- *`RPC`*: task sent a Remote Procedure Call and the receiver has not got it yet.
- *`RETURN`*: task waits for reply to a Remote Procedure Call.

The returned task state is just an approximative information. Timer and other hardware interrupts may cause a change in the state of the queried task before the caller could evaluate the returned value. The caller should disable interrupts if it wants reliable info about another task.

A task can find its own task data structure with:

```
RT_TASK *rt_whoami (void);
```

Tasks can choose whether or not to save floating point registers at context switches:

```
int rt_task_use_fpu (RT_TASK* task, int use_fpu_flag);
   // informs the scheduler that floating point arithmetic
   // operations will be used by the "task".
   // If "use_fpu_flag" has a nonzero value, FPU context is also
   // switched when task or the kernel become active. This makes task
   // switching slower. The initial value of this flag is set by
   // "rt_task_init()" when the real time task is created. By default,
   // a Linux "task" has this flag cleared. It can be set with the
   // "LinuxFpu" command line parameter of the "rtai_sched" module.

void rt_linux_use_fpu (int use_fpu_flag);
   // informs the scheduler that floating point arithmetic
   // operations will be used in the background task (i.e.,
   // the Linux kernel itself and all of its processes).
```

8.2.7. Getting the time

RTAI provides several function calls for getting the current time (`ABCscheduler/rtai_sched.c`):

```
RTIME rt_get_time(void)
RTIME rt_get_time_cpuid(unsigned int cpuid)
RTIME rt_get_time_ns(void)
RTIME rt_get_time_ns_cpuid(unsigned int cpuid)
RTIME rt_get_cpu_time_ns(void)
```

The time is given in "ticks", or in nanoseconds. The parameter *`cpuid`* indicates the number of the CPU in a multi-processor system, and these calls (and `rt_get_cpu_time_ns`) read the local *Time Stamp Clock*, instead of the external timer chip. The latter has usually a lower resolution.

8.3. Interrupts and traps

An interrupt handler (Section 3.3) must be registered with the operating system via a call of the following function:

```
int rt_request_global_irq (
   unsigned int irq,
   void (*handler)(void)
);
```

This call installs the function *`handler`* as the interrupt service routine for IRQ level *`irq`*. *`handler`* is then invoked whenever interrupt number *`irq`* occurs. The installed handler must take care of properly activating any Linux handler using the same irq number, by calling the `void rt_pend_linux_irq (unsigned int irq)` function, which "pends" the interrupt to Linux (in software!). This means that Linux will process the interrupt as soon as it gets control back from RTAI. Note that, at that time, hardware interrupts are again *enabled* for RTAI. The use of `rt_pend_linux_irq()` does only make sense for *edge-triggered* interrupts (Section 3.2): the level-triggered one is still active, unless you have acknowledged it already *explicitly*.

From an RTAI task, one can also register an interrupt handler with Linux, via

```
int rt_request_linux_irq (
   unsigned int irq,
   void (*handler)(int irq, void *dev_id, struct pt_regs *regs),
   char *linux_handler_id,
   void *dev_id
);
```

This forces Linux to share the interrupt. The handler is appended to any already existing Linux handler for the same irq and run as a Linux irq handler. The handler appears in `/proc/interrupts`, under the name given in the parameter *`linux_handler_id`*. The parameter *`dev_id`* is passed to the interrupt handler, in the same way as the standard Linux irq request call.

```
void rt_request_timer (
   void (*handler)(void),
   int tick,
   int apic
);
```

registers the *`handler`* as the ISR of a timer interrupt. If *`tick`* is zero, the timer is executed only once. If *`apic`* is nonzero, the local APIC is used (Section 3.2). The difference with the timer tasklets (Section 8.2.3) is that the latter are not directly registered as an interrupt handler, but executed by a timer manager (which is itself woken up by a timer).

Floating point register saving is *on by default* in RTAI interrupt handlers. The DSR functionality (Section 3.4) is available through tasklets, and ASR functionality through the `signal()` parameter. One can also select which CPU must receive and handle a particular IRQ, via the `rt_assign_irq_to_cpu(int irq, int cpu)` function. `rt_reset_irq_to_sym_mode(int irq)` resets this choice, back to the symmetric "don't care" behaviour.

In RTAI, application programmers must explicitly enable interrupts themselves, via `rt_irq_enable()`. Whether this is done in the ISR or in the DSR depends on the hardware of the application: if it has an interrupt ready immediately, enabling the interrupts in the ISR could cause recursive calls to the ISR, possibly blocking the system.

Section 3.3 discussed the concept of *traps* and *trap handlers* . The API that RTAI offers is as follows:

```
   // data structure of handler
typedef int (*RT_TRAP_HANDLER)(
  int,                            // interrupt vec
  int,                            // signal number
  struct pt_regs *,               // argument pointers that can be
                                  // given to a trap handler (see Linux)
  void *                          // data pointer
);

   // fill in trap handler data structure:
int rt_trap_handler(
  int vec,
  int signo,
  struct pt_regs *regs,
  void *dummy_data
);

   // register trap handler:
RT_TRAP_HANDLER rt_set_task_trap_handler(
  RT_TASK *task,          // task which registers handler
  unsigned int vec,       // interrupt vec which triggers handler
  RT_TRAP_HANDLER handler // data structure of handler
);
```

RTAI reserves 32 *system signals*, most of them correspond to what standard Linux uses. These signals are denoted by "*`signo`*" in the code above, and are defined in the data structure *`rtai_signr[NR_TRAPS]`* in the file `"arch/i386/rtai.c`, for i386 only. The default configuration policies of RTAI are: (i) to add the same handler to all traps, (ii) to trap the non-maskable interrupt of the processor and let it do nothing (getting it in the first place indicates that something major has gone wrong), and (iii) to suspend a task that calls a non-existing handler.

8.4. IPC: synchronization

Also in this area, RTAI offers the whole range of synchronization primitives: semaphore and mutex, condition variable, and barrier or flags ("bits").

8.4.1. Semaphore and mutex

RTAI has counting semaphores, binary semaphores and recursive semaphores , Section 4.6.1. semaphores can block tasks waiting on them in FIFO or priority order;

Semaphores in RTAI have the following API:

```
 void rt_sem_init      (SEM* sem, int value);
 int rt_sem_signal     (SEM* sem);
 int rt_sem_wait       (SEM* sem);
   // version that returns immediately when not free:
 int rt_sem_wait_if    (SEM* sem);
   // versions with a timeout:
 int rt_sem_wait_until (SEM* sem, RTIME time);  // absolute time
 int rt_sem_wait_timed (SEM* sem, RTIME delay); // relative time
```

RTAI semaphores have *priority inheritance*. and (adaptive) *priority ceiling* (Section 4.9).

8.4.2. POSIX mutex

RTAI implements the standard POSIX mutexes (Section 4.6.2), with the prescribed *priority inheritance*. The API is, of course, the standard POSIX API as presented in Section 4.6.2.

8.4.3. Spinlocks

Application programmers can choose from a wide variety of spinlocks, each with well-defined scope. Basically, they look like the spinlocks in Linux, with a “`rt_`” prefix, but using the same data structures. But the RTAI spinlocks need an extra level with respect to Linux, because Linux runs on an hardware simulation layer as soon as RTAI has been activated. Indeed, from that moment on, the Linux calls are replaced by “soft” versions, in the sense that RTAI can always pre-empt critical Linux sections. Here is the list of RTAI spinlocks:

```
unsigned long flags;
spinlock_t lock;

rt_spin_lock(&lock);
 /* critical section in Linux (as the ‘spin_lock()’ there, hence
    Linux’s (soft) interrupts still pass), but pre-emptable by RTAI.
 */
rt_spin_unlock(&lock);

rt_spin_lock_irq(&lock);
 /* same as above but Linux’s soft interrupts disabled. */
rt_spin_unlock_irq(&lock);

flags = rt_spin_lock_irqsave(&lock);
 /* critical section in RTAI with hardware interrupts disabled
```

```
    on current CPU. */
rt_spin_lock_irqrestore(flags,&lock);
```

The following locks don't need a lock data structure, because they are drastic, and use a "global lock" over all processors:

```
rt_global_cli();
 /* critical section with interrupts disabled on the calling CPU,
    and "global lock" for all CPUs. */
rt_global_sti();

flags = rt_global_save_flags_and_cli();
 /* as "rt_global_cli()", but saves the state of the interrupt flag,
    and the "global lock" flag. */
rt_global_restore_flags(flags);

flags = hard_lock_all();
 /* Most drastic way of making the system safe from pre-emption by
    interrupts.
    On UP boxes is the same as "rt_global_save_flags_and_cli()"
    above. On SMP locks out all the other CPUs, sending then an
    IPI (inter-processor interrupt) signal. */
hard_unlock_all(flags);
```

The normal Linux spinlocks still work in RTAI, so be careful when using them, because they won't always offer the same protection in RTAI hard real ime as what you expect from knowing how they behave in un-modified Linux.

8.4.4. Condition variable

RTAI implements the standard POSIX condition variables (Section 4.7).

8.4.5. Barrier/flags

RTAI has a barrier-like (Section 4.6.5) primitive, which it calls *bits*. It allows tasks to suspend on an AND or OR combination of bits sets in a 32 bit mask called "BITS" (`include/rtai_bits.h`):

```
struct rt_bits_struct {
   struct rt_queue queue;  // must be first in struct
   int magic;
   int type;  // needed because BITS and semaphores share some things
   unsigned long mask;
};

typedef struct rt_bits_struct BITS;
```

Tasks can read and write bits in this mask, and perform "wait" calls on the mask. The full API: is as follows:

```
#include <rtai_bits.h>

   // basic bit operation functions, indicated by macros:
#define SET_BITS              0
#define CLR_BITS              1
#define SET_CLR_BITS          2
#define NOP_BITS              3

void rt_bits_init(BITS *bits, unsigned long mask)
   // create and initialize the bits structure pointed to by "bits",
   // setting bits mask to "mask".

int rt_bits_delete(BITS *bits)
   // delete the "bits" data structure

unsigned long rt_get_bits(BITS *bits)
   // get the actual value of the "bits" mask.

unsigned long rt_bits_signal(
  BITS *bits,
  int setfun,
  unsigned long masks)
   // execute "setfun" (which is any of the basic bits operations
   //  above: SET_BITS, etc.), oring/anding masks onto the actual
   // bits mask, schedule any task blocked on "bits" if the new bits
   // mask meets its request;
   // returns the value of bits after executing setfun;
   // in case of combined operations (AND and OR), "masks" is to be
   // cast to a pointer of a two elements array of unsigned longs
   // containing the masks to be used for the combined "setfun".

int rt_bits_reset(BITS *bits, unsigned long mask)
   // unconditionally schedule any task blocked on "bits" and
   // reset its mask to "mask";
   // returns the value of bits mask before being reset to "mask".

int rt_bits_wait(
  BITS *bits,
  int testfun,
  unsigned long testmasks,
  int exitfun,
  unsigned long exitmasks,
  unsigned long *resulting_mask)
   // test "bits" mask against "testmasks" according to "testfun"
   // (which is any of the test functions above, e.g., SET_BIT, etc.);
   // if the test is not satisfied block the task;
   // whenever the condition is met, execute "exitfun:, and any bits
   // operation above, using "exitmasks",
   // save the the mask resulting after the whole processing in the
   // variable pointed by "resulting_mask".

int rt_bits_wait_if(
  BITS *bits,
  int testfun,
  unsigned long testmasks,
  int exitfun,
  unsigned long exitmasks,
  unsigned long *resulting_mask)
   // as "rt_bits_wait",
   // but does not block if "testfun" is not satisfied.

int rt_bits_wait_until(
  BITS *bits,
  int testfun,
  unsigned long
  testmasks,
  int exitfun,
  unsigned long exitmasks,
  RTIME time,
  unsigned long *resulting_mask)
   // as "rt_bits_wait",
   //  but waits at most till "time" expires.

unsigned long rt_bits_wait_timed(
  BITS *bits,
  int testfun,
  unsigned long testmasks,
  int exitfun,
  unsigned long exitmasks,
  RTIME delay,
  unsigned long *resulting_mask)
   // as "rt_bits_wait_until",
   // but waits at most for "delay" to meet the required condition.
```

8.5. IPC: data exchange.

RTAI has messages, mailboxes, and POSIX message queues (“pqueues”), including synchronous message passing semantics (Section 5.3), FIFOs, Remote Procedure Calls, and shared memory.

8.5.1. Messages

RTAI makes the distinction between messages and mailboxes, as explained in Section 5.3. The messages are the more primitive form, and in RTAI, the basic implementation of messages carry only a *four byte* message in the call itself. So, no buffering must be provided. The API for this simple inter-task messaging is:

```
RT_TASK* rt_send (RT_TASK* task, unsigned int msg);
  // sends the message "msg" to the task "task".
  // If the receiver task is ready to get the message,
  // "rt_send" returns immediately.
  // Otherwise the caller task is blocked.

RT_TASK* rt_send_if (RT_TASK* task, unsigned int msg);
  // sends the message “if possible”. If the receiver task is not
  // ready, the sending task just continues.
  // On success, "task" (the pointer to the task that received the
  // message) is returned.
  // If message has not been sent, 0 is returned.

RT_TASK* rt_send_until (RT_TASK* task, unsigned int msg, RTIME time);
RT_TASK* rt_send_timed (RT_TASK* task, unsigned int msg, RTIME delay);
  // As "rt_send", but the sending is given up after either an
  // absolute "time", or a relative "delay".

RT_TASK* rt_receive (RT_TASK* task, unsigned int *msg);
  // gets a message from the "task", and stores it in the buffer "msg"
  // that the caller task provides.
  // If "task" is equal to 0, the caller accepts messages from any
  // task. If there is a pending message, "rt_receive" returns
  // immediately. Otherwise the caller task is blocked and queued up.

RT_TASK* rt_receive_if (RT_TASK* task, unsigned int *msg);
  // as "rt_receive", but only “if possible”.

RT_TASK* rt_receive_until (RT_TASK* task, unsigned int *msg, RTIME time);
RT_TASK* rt_receive_timed (RT_TASK* task, unsigned int *msg, RTIME delay);
  // as "rt_receive", but with time limits as in the send calls.
```

Blocking may happen in priority order or on a FIFO base. This is determined by an RTAI compile time option *MSG_PRIORD*.)

More recently, RTAI got so-called *extended messages*. These are less efficient than their four-byte cousins, but more flexible in that they allow messages of arbitrary size. To this end, the extended message functions use a double buffer data structure:

```
struct mcb_t {
  void *sbuf;  // buffer for the sender
  int sbytes;  // number of bytes sent
  void *rbuf;  // buffer for the receiver
  int rbytes;  // number of bytes received
};
```

The following function prototypes are quite self-explanatory, with *smsg* indicating the sender’s message buffer, *ssize* the sender’s message size, and *rmsg* and *rsize* similarly for the receiver.

```
RT_TASK *rt_sendx(RT_TASK *task, void *smsg, int ssize)

RT_TASK *rt_sendx_if(RT_TASK *task, void *smsg, int ssize)

RT_TASK *rt_sendx_until(
  RT_TASK *task,
  void *smsg,
  int ssize,
  RTIME time)

RT_TASK *rt_sendx_timed(
  RT_TASK *task,
  void *smsg,
  int ssize,
  RTIME delay)

RT_TASK *rt_receivex(
  RT_TASK *task,
  void *msg,
  int size,
  int *truesize)

RT_TASK *rt_receivex_if(
  RT_TASK *task,
  void *msg,
  int size,
  int *truesize)

RT_TASK *rt_receivex_until(
  RT_TASK *task,
  void *msg,
  int size,
  int *truesize,
  RTIME time)

RT_TASK *rt_receivex_timed(
  RT_TASK *task,
  void *msg,
  int size,
  int *truesize,
  RTIME delay)

RT_TASK *rt_rpcx(
  RT_TASK *task,
  void *smsg,
  void *rmsg,
  int ssize,
  int rsize)

RT_TASK *rt_rpcx_if(
  RT_TASK *task,
  void *smsg,
  void *rmsg,
  int ssize,
  int rsize)

RT_TASK *rt_rpcx_until(
  RT_TASK *task,
  void *smsg,
  void *rmsg,
  int ssize,
  int rsize,
  RTIME time)

RT_TASK *rt_rpcx_timed(
  RT_TASK *task,
  void *smsg,
  void *rmsg,
  int ssize,
  int rsize,
  RTIME delay)

T_TASK *rt_returnx(RT_TASK *task, void *msg, int size)
   // ???

int rt_isrpcx(RT_TASK *task)
   // ???
```

8.5.2. Mailboxes

RTAI supports mailboxes (Section 5.3). They are flexible in the sense that they allow to send any message size by using any mailbox buffer size. The original implementation uses a FIFO (First In, First Out) policy; a recent addition are “typed” mailboxes, that have a priority message delivery option. Sending and receiving messages can be done with several policies:

- *Unconditionally*: the task blocks until the whole message has passed.
- *Best-effort*: only pass the bytes that can be passed immediately.
- *Conditional on availability*: only pass a message if the whole message can be passed immediately.
- *Timed*: with absolute or relative time-outs.

The API for mailboxes is given in `include/rtai_sched.h` (of all places. . .):

```
struct rt_mailbox {
   int magic;    // identifier for mailbox data structure
   SEM sndsem,   // semaphores to queue sending...
       rcvsem;   // ... and receiving tasks.
   RT_TASK *waiting_task, // pointer to waiting tasks
           *owndby;       // pointer to task that created mailbox
   char *bufadr;          // mailbox buffer
   int size,     // mailbox size
       fbyte,    // circular buffer first byte pointer
       lbyte,    // circular buffer last byte pointer
       avbs,     // bytes in buffer
       frbs;     // bytes free
   spinlock_t lock;       // lock to protect access to buffer
};
```

```
typedef struct rt_mailbox MBX;

int rt_typed_mbx_init(MBX *mbx, int size, int qtype);
   // Initialize a mailbox "mbx" with a buffer of "size" bytes,
   // queueing tasks according to the specified type: FIFO_Q, PRIO_Q and
   // RES_Q.

int rt_mbx_init(MBX *mbx, int size);
   // equivalent to rt_typed_mbx_init(mbx, size, PRIO_Q)

int rt_mbx_delete(MBX *mbx);
   // Delete the mailbox "mbx".

int rt_mbx_send(MBX *mbx, void *msg, int msg_size);
   // Send unconditionally, i.e. return when the whole message has
   // been received or an error occured, to the mailbox "mbx", the
   // message pointed by "msg", whose size is "msg_size" bytes.
   // Returns the number of unsent bytes.

int rt_mbx_send_wp(MBX *mbx, void *msg, int msg_size);
   // As "rt_mbx_send", but only available bytes.
   // “_wp” stands for: “what possible.”

int rt_mbx_send_if(MBX *mbx, void *msg, int msg_size);
   // Send to the mailbox "mbx" only if all "msg_size" bytes
   // of "msg" can be received immediately.
   // Returns the number of unsent bytes, i.e. either 0 or "msg_size".
   // “_if” stands for: “if available.”

int rt_mbx_send_until(MBX *mbx, void *msg, int msg_size, RTIME time);
   // As "rt_mbx_send", unless the absolute time dead-line "time"
   // is reached.

int rt_mbx_send_timed(MBX *mbx, void *msg, int msg_size, RTIME delay);
   // As "rt_mbx_send", unless the time-out "delay" has expired.

   // Similar semantics for receiving message:
int rt_mbx_receive(MBX *mbx, void *msg, int msg_size);
int rt_mbx_receive_wp(MBX *mbx, void *msg, int msg_size);
int rt_mbx_receive_if(MBX *mbx, void *msg, int msg_size);
int rt_mbx_receive_until(MBX *mbx, void *msg, int msg_size, RTIME time);
int rt_mbx_receive_timed(MBX *mbx, void *msg, int msg_size, RTIME delay);

int rt_mbx_evdrp(MBX *mbx, void *msg, int msg_size);
   // This is the “unsafe” version, that doesn’t protect against
   // overwriting the circular message buffer.
   // The name stands for “eventual dropping” of data. (???)
```

Typed mailboxes offer a functionality that is a *superset* of the mailboxes above, adding the following features:

- *Message broadcasting*: a message is sent to *all* tasks that are pending on the same mailbox.
- *Priority configuration*: a *urgent* or *normal* wakeup policy can be set when creating the mailbox.

These features are achieved by adding a 1-byte *type field* to every message inserted in a typed mailbox. So, when receiving it is possible to discriminate normal, urgent and broadcast messages. The type field is silently removed by the receiving functions, so from the user point of view it is not visible. Users must consider type fields only when specifying the types mailbox sizes.

The API for typed mailboxes is given in `include/rtai_tbx.h`:

```
struct rt_typed_mailbox {
   int magic;
   int waiting_nr;     // number of tasks waiting for a broadcast
   SEM sndsmx,         // semaphores to queue sending...
       rcvsmx;         // ... and receiving tasks.
   SEM bcbsmx;         // binary semaphore needed to wakeup the
                       // sleeping tasks when the broadcasting of a
                       // message is terminated
   RT_TASK *waiting_task;
   char *bufadr;       // mailbox buffer
   char *bcbadr;       // broadcasting buffer
   int size;           // mailbox size
   int fbyte;          // circular buffer read pointer
   int avbs;           // bytes occupied
   int frbs;           // bytes free
   spinlock_t buflock; // lock to protect buffer access
};

typedef struct rt_typed_mailbox TBX;

   // The function prototypes are similar to normal mailboxes,
   // with "_mbx_" replaced by "_tbx_". For example:
int rt_tbx_init(TBX *tbx, int size, int type);
int rt_tbx_send(TBX *tbx, void *msg, int msg_size)
   // etc.
   // Some functions are new:
int rt_tbx_broadcast(TBX *tbx, void *msg, int msg_size);
int rt_tbx_broadcast_if(TBX *tbx, void *msg, int msg_size);
int rt_tbx_broadcast_until(TBX *tbx, void *msg, int msg_size, RTIME time);
int rt_tbx_broadcast_timed(TBX *tbx, void *msg, int msg_size, RTIME delay);

int rt_tbx_urgent(TBX *tbx, void *msg, int msg_size);
int rt_tbx_urgent_if(TBX *tbx, void *msg, int msg_size);
int rt_tbx_urgent_until(TBX *tbx, void *msg, int msg_size, RTIME time);
int rt_tbx_urgent_timed(TBX *tbx, void *msg, int msg_size, RTIME delay);
```

The *unconditional* versions of mailbox communication correspond to *synchronous message passing*.

8.5.3. POSIX message queues

RTAI supports standard POSIX message queues (Section 5.3).

8.5.4. FIFO

FIFOs are a basic IPC data exchange primitive, and well supported under RTAI. It offers an API for kernel space FIFOs, and one for user space FIFOs:

```
struct rt_fifo_info_struct{
   unsigned int fifo_number;
   unsigned int size;
   unsigned int opncnt;
   char name[RTF_NAMELEN+1];
};

struct rt_fifo_get_info_struct{
   unsigned int fifo;
   unsigned int n;
   struct rt_fifo_info_struct *ptr;
};

   // initialize FIFO data structure:
int rtf_init(void);

/* Attach a handler to an RT-FIFO.
 *
 * Allow function handler to be called when a user process reads or
 * writes to
 * the FIFO. When the function is called, it is passed the fifo number
 * as the
 * argument.
 */

extern int rtf_create_handler(unsigned int fifo,   /* RT-FIFO */
      int (*handler)(unsigned int fifo)   /* function to be called
*/);
```

Here is the skeleton of a user space task and a hard real-time task, that use a FIFO to communicate; the other IPC primitives use similar skeletons.

```
// user space task:
int main(int argc,char *argv[])
{
  int rtf, cmd;
  int data[...];
  double ddata[...];
  ...
  if ((rtf = open("/dev/rtf0", O_RDONLY)) < 0) {          ❶
          fprintf(stderr, "Error opening /dev/rtf0\n");
          exit(1);
  }
  if ((cmd = open("/dev/rtf1", O_WRONLY)) < 0) {          ❷
          fprintf(stderr, "Error opening /dev/rtf1\n");
          exit(1);
  }
  while (...) {                                           ❸
    write(cmd, &data, ...);
    ...                                                   ❹
    read(rtf, &ddata, ...);
    ...
  };
  ...
  return 0;
}

// module that creates hard real-time task:
#define RTF 0
#define CMD 1

static RT_TASK mytask;

int init_module(void)
{
        rtf_create(RTF, 4000);                            ❺
        rtf_create(CMD, 100);                             ❻
        rt_task_init(&mytask, fun, 0, STACK_SIZE, 0, 1, 0);
        rt_set_runnable_on_cpus(&mytask, ...);
        rt_assign_irq_to_cpu(TIMER_8254_IRQ, TIMER_TO_CPU);
        rt_linux_use_fpu(1);
        now = rt_get_time();
        rt_task_make_periodic(&mytask, now + 2000, ...);
        return 0;
}

// function run in real-time task:
static void fun(int t) {
  ...
  while (...) {
    cpu_used[hard_cpu_id()]++;
    rtf_put(RTF, ..., ...);
    rtf_get(CMD, ..., ...):
    rt_task_wait_period();
  }
}
```

❶ Opens first FIFO as a user space device.

❷ Opens second FIFO as a user space device.

❸ Writes data in the FIFO.

❹ Reads data from the FIFO.

One can add a handler to a FIFO, via `rtf_create_handler()`. One can also send a signal to notify data availability, via `rtf_set_async_sig(int fd, int signum)`. This handler and signal functionality is not available for the other IPC primitives.

8.5.5. RPC

RTAI supports Remote Procedure Calls, Section 5.3. (Even over a network, In which case the user is responsible for using appropriate hardware, of course. This text skips the details of this latter functionality, because it falls outside of the scope of hard real-time systems.) The on-system RPC in RTAI works as a “send/receive” message pair: a task sends a four-byte message to another task, and then waits until a reply is received. The caller task is always blocked and queued up. Calling this a “Remote Procedure Call” is a bit ambitious: the communicating tasks just send four bytes, and they have to agree on a protocol that defines the *meaning* of these four bytes, and whether or not the message triggers the execution of a procedure call at the receiver’s end. The API for this form of RPC is:

```
RT_TASK *rt_rpc(
  RT_TASK *task,
  unsigned int to_do,
  unsigned int *reply);
   // The receiver task may get the message with any "rt_receive_*"
   // function. It can send the answer with "rt_return()".
   // "reply" points to a buffer provided by the caller.

RT_TASK *rt_return(
  RT_TASK *task,
  unsigned int reply);

RT_TASK *rt_rpc_if(
  RT_TASK *task,
  unsigned int to_do,
  unsigned int *result);

RT_TASK *rt_rpc_until(
  RT_TASK *task,
  unsigned int to_do,
  unsigned int *result,
  RTIME time);

RT_TASK *rt_rpc_timed(
  RT_TASK *task,
  unsigned int to_do,
  unsigned int *result,
  RTIME delay);

int rt_isrpc(RT_TASK *task);
```

```
// After receiving a message, by calling "rt_isrpc" a task
// can find out whether the sender task "task" is waiting for
// a reply or not.
// "rt_return" is intelligent enough to not send an answer to
// a task which is not waiting for it. Therefore using "rt_isrpc"
// is not  necessary and discouraged.
```

The meaning of the suffixes “*`_if`*”, “*`_until`*”, and “*`_timed`*” is as in the APIs of messages and mailboxes.

8.6. Memory management

Shared memory implementation in `shmem`. Again symmetric. Dynamic memory management;

(TODO: more details.)

8.7. Real-time device drivers

spdrv, rtnet, plus strong integration with Comedi.

(TODO: more details.)

8.8. `/proc` interface

The `/proc` interface is an extension to the standard Linux `/proc` interface feature: files under the subdirectory `/proc/rtai` give status and debug information of the currently active RTAI modules. These files are activated when the associated module is inserted into the kernel.

`/proc` interface code can be found in most RTAI source files. It’s a non real-time feature (hence, only to be used by normal user space tasks), but it requires support from the real-time kernel; this support is implemented again via *traps*.

8.9. RTAI loadable modules

RTAI’s functionality is made available by dynamically *loading modules* into the running (and patched) Linux kernel. Every module extends the API of the kernel with some new “objects” (i.e., function calls and data structures). Not all modules are needed in all cases, but, vice versa, dependencies exist between modules, i.e., in order to use functionality in one module, one often also needs to load other modules first.

`rtai` core module `rtai.c`, and made in `rtaidir`.

Scheduler module `ABCscheduler/rtai_sched.c`.

Tasklet module: allocates and initializes the data structures for the tasklet and timer queues; starts the `timers_manager` task, that is responsible for the execution of the timers;

Scheduler module `....`

Extra scheduler module `....`

RTAI utilities module `....`

Types mailboxes module `....`

pthreads module `....`

Memory manager module `....`

FIFOs module `fifos/rtai_fifos.c`.

LX/RT module `lxrt/lxrt.c`.

Serial line module `spdrv/rtai_spdrv.c`.

C++ module `....`

Network RPC module `net_rpc/net_rpc.c`.

Tracing module `trace/rtai_trace.c`.

Watchdog module `watchdog/rtai_watchdog.c`.

Bits module `bits/rtai_bits.c`.

(TODO: explain contents of the different RTAI modules; dependencies: what must be loaded in order to use the different functionalities mentioned above?)

8.10. Specific features

RTAI has developed a number of features that common real-time operating systems miss:

- LX/RT is the component that allows user space tasks to execute soft and hard real-time functions. Because this feature is quite extensive, section Section 11.5 gives more details.
- Dynamic memory allocation, also by real-time tasks. (TODO: give details.)
- Integration of the Linux Trace Toolkit (http://www.opersys.com/LTT/index.html), which allows to trace (i.e., log to a buffer) a large number of activities from the kernel: interrupts, scheduling, creation of tasks, etc. (TODO: give details.)
- C++ support, Chapter 12.

Chapter 9. Linux-based real-time and embedded operating systems

This Chapter presents “spin-offs” of the standard Linux kernel that provide hard real-time performance, or that are targeted to embedded use.

9.1. Introduction

There are two major developments at the RTOS level: RTLinux and RTAI. RTAI forked off an earlier version of RTLinux. RTLinux and RTAI do basically the same thing (and do it with industrial strenght quality, except maybe for documentation. . .), they make their sources available, they have partial POSIX compliance, but they don’t use compatible APIs. In the *embedded* (but non real-time) Linux world, projects have emerged, such as uCLinux, and Etlinux. But probably standard Linux is the major workhorse here, thanks to its great configurability.

9.2. RTLinux: Real-Time Linux

RTLinux (http://www.rtlinux.com) is a patch for the standard Linux kernel (often called the “vanilla” Linux kernel), for single as well as for multi-processor kernels. It offers all components of a hard real-time system in a multi-threaded real-time kernel, in which standard Linux is the lowest-priority thread. One advantage (or disadvantage, depending on your taste) of this approach is that real-time space and Linux space (both kernel space and user space) are strictly separated: programmers have to specify explicitly which of their tasks should run with real-time capabilities, and which others should not. This separation also relieves the real-time kernel from “bookkeeping” tasks such as booting, device initialization, module (un)loading, or dynamic memory allocation. None of these have real-time constraints, hence they naturally belong to Linux and not RTLinux. From programming point of view, most, but not all, functionality, habits and tools of Linux remain available at no cost, *and* the real-time application can run and be debugged on the same computer on which it is developed, without the need for cross-compilation tools. This makes “migration” for Linux users quite painless.

The disadvantage of a distribution in the form of a kernel patch is that this patch has (i) to be maintained (by the RTLinux developers) over evolving kernel versions, and (ii) applied (by the users) each time they upgrade their kernel. Pre-patched versions of some kernel versions are available from the RTLinux web page. The RTLinux patch is minor: it provides a “virtual interrupt” emulation to standard Linux, and offers a kernel space micro-kernel with real-time scheduled threads. RTLinux intercepts all hardware interrupts, checks whether an interrupt is destined for a real-time service routine (and launches the corresponding ISR if it is), or forwards them to Linux in the form of a virtual interrupt, which is held until no real-time activity must run. In this scheme, Linux is never able to disable *hardware* interrupts.

RTLinux comes (after compilation) as a set of loadable modules within Linux: the core module with the above-mentioned interrupt controller handler, a real-time scheduler (with static priorities), a timer module, a FIFO implementation, shared memory and most real-time lock and event primitives. This modularity makes customization easier, and increases the embeddability (because unnecessary modules need not be loaded).

9.2.1. Functionalities

RTLinux offers basic POSIX compliance: it has implemented the *Minimal Realtime System Profile* (POSIX 1001.13, PSE51). This means that it has basic thread management, IPC primitives, and `open`/`read`/ `write`/`...` function calls, but only for basic *device* I/O rather than full *file system* support. RTLinux has support for mutexes, condition variables, semaphores, signals, spinlocks, and FIFOs. It implements some form of *user space real time*, based on the signal mechanism. RTLinux tasks can communicate with Linux tasks, with the guarantee that this IPC is *never* blocking at the RTLinux side.

Some function calls do not follow the POSIX standard; these are named `pthread_..._np()`, where the “`np`” stands for “non-portable.” This behaviour of adding “`pthread_..._np()` functions in a POSIX-compatible operating system is explicitly allowed by the POSIX standard. RTLinux uses this behaviour, but none of its core functionality depends on it.

9.2.2. MiniRTL

miniRTL (http://www.thinkingnerds.com/projects/minirtl/minirtl.html) is a (not actively maintained) sub-project of RTLinux that offers a small-sized real-time Linux that is small enough to boot from a single floppy (or small Flash memory device) into a ramdisk, yet offers the most important features of Linux. miniRTL is intended to be useful as the basis for embedded systems, but also provides a means for real-time “newbies” (or non-Linux users) to learn more about real-time Linux.

9.2.3. The RTLinux patent

RTLinux has matured significantly over three major versions of RTLinux, and, since the 3.0 release, not many API changes have occurred. This is partially due to the carefully conservative policy of RTLinux maintainer Victor Yodaiken, but partially also to the fact that RTLinux started with a closed-source, proprietary, patent-protected version. That means that there are two branches of RTLinux: RTLinux/GPL (free software), and RTLinux/PRO (non-free software, where most of the developments and hardware ports are taking place). The start of such a closed-cource branch was possible, because Yodaiken didn’t include contributions in the RTLinux core with (GPL) copyrights of other contributors than FSMLabs. This move was not too well appreciated in the free software community, but was practically inevitable in order to build a business around RTLinux development. The support for, and response to, users of the GPL-ed version has drastically been reduced.

The RTLinux approach is covered by US Patent 5995745 (http://www.patents.ibm.com/details?pn=US05995745__), issued on November 30, 1999. RTLinux comes with a remarkable license for using this patent (see the file `PATENT` in the source distribution of RTLinux). The following is an excerpt from that patent license file:

> ...
>
> The Patented Process may be used, without any payment of a royalty, with two (2) types of software. The first type is software that operates under the terms of a GPL (as defined later in this License). The second type is software operating under Finite State Machine Labs Open RTLinux (as defined below). As long as the Licensee complies with the terms and conditions of this License and, where applicable, with the terms of the GPL, the Licensee may continue to use the Patented Process without paying a royalty for its use.
>
> ...
>
> —THE OPEN RTLINUX PATENT LICENSE

With this patent, FSMLabs tries to find a balance between stimulating development under the GPL on the one hand, and generating a business income from real-time operating system development and service on the other hand. This patent is (at the time of this writing) not valid outside of the USA. FSMLabs has expressed its intention to *enforce* the patent, which has led to very strong reactions in the free software community. One of these reactions has been the development of an alternative approach, free of the patent claims (see Section 10.1); another reaction is the massive transition of community development efforts towards RTAI.

9.3. RTAI: the Real-Time Application Interface

RTAI (http://www.rtai.org) has its origin in RTLinux, when main developer Paolo Mantegazza wanted to bring his work and experiences with real-time on DOS to Linux. The “schism” from RTLinux that gave birth to RTAI occurred quite early on in the history of RTLinux, when Mantegazza wanted some features for his own work (e.g., multi-processor support) that did not exist in RTLinux, and in which the RTLinux developers showed no interest. The APIs of RTLinux and RTAI are similar (both are RTOSs anyway), but not trivially exchangeable. And they become even more and more distinct over time. They do, however, support about the same set of POSIX primitives.

RTAI is more of a “bazaar”-like project than RTLinux, in the sense that it happily accepts contributions from anybody, without sticking to a strict design vision, or code tree and documentation discipline. In that sense it responds better to user requests, evolves rapidly, but possibly at the price of giving a chaotic impression to new users. Anyway, it has succeeded in attracking almost all community development efforts in the area of real-time for Linux, at the expense of the RTLinux project.

This document takes RTAI as an example RTOS to investigate in more technical details in Chapter 8. The following sections give some information about an important non-technical aspect of RTAI: its relationship with the RTLinux patent (Section 9.2.3).

9.3.1. RTAI and the RTLinux patent

RTLinux’s owner FSMLabs has done little to clear up the uncertainty around the legal repercussions of its patent, which could scare away potential commercial interest in RTAI. However, the RTAI community has been able to clear up matters, in different ways:

1. The license of the RTAI core changed from LGPL to GPL, so that it complies with the patent.
2. Eben Moglen, Professor of law at Columbia University, and legal adviser to the Free Software Foundation (http://www.fsf.org), published a legal study, that concludes that the patent is not enforceable on applications done with RTAI. A summary of his study (http://www.aero.polimi.it/~rtai/documentation/articles/moglen.html) can be read at the RTAI homepage.
3. Karim Yaghmour's rebuttal of the RTLinux patent rights. Basically, the patent was submitted too long after the patented ideas were published and available in code form. The details can be found in his "Check Mate" (http://www2.fsmlabs.com/mailing_list/rtl.w5archive/advocacy-0204/msg00042.html) posting on the RTLinux advocacy mailing list.
4. *Adeos.* (See Section 10.1 for more technical detail.) This is a nano-kernel, that offers an alternative to the patented concept of RTLinux. At the time of writing, Adeos has not yet been accepted as the real core of RTAI, but several positive testing and porting signs emerge from the community.
5. RTAI has introduced additions to normal Linux task management and scheduling, that offer the functionality to schedule *user space* tasks with hard real-time determinism (Section 11.5). And Linux user space applications are not within the scope of the patent's claims.

9.4. uCLinux

uCLinux (http://www.uclinux.org): for MMU-less processors; small footprint (about 500 to 900 kB); full TCP/IP stack; support for various file systems. Has real-time functionality too. An introduction to uCLinux can be found here (http://www.snapgear.com/tb20020807.html).

9.5. Etlinux

Etlinux (http://www.prosa.it/etlinux/) is a complete Linux-based system designed to run on very small industrial computers, such as i386 and PC/104 modules with not more than 2 Megabytes of RAM.

Chapter 10. Non-Linux real-time operating systems

There are many application areas where using a Linux kernel is not a good idea, because of memory footprint, feature bloat, licensing and patent issues, processor support, etc. Moreover, Linux was certainly not the first free software operating system, particularly not in the area of real-time. This chapter points out some of the non-Linux alternatives that are available under free software licenses. From them, eCos has probably been the most successful in gathering a large user and development community.

10.1. The Adeos nano-kernel

The *Adaptive Domain Environment for Operating Systems* (Adeos ()) is not really an (RT)OS in itself, but a software layer between the hardware interrupts and the operating system. Or rather, between the hardware and the *various* operating systems that can run on top of it. Indeed, Adeos is capable of “hosting” more than one OS on top of it, and these OSs don’t know about each other, as long as they ask Adeos to pass through the interrupts they need.

The Adeos design was done by Karim Yaghmour, because he wanted to find a way to avoid the FSMLabs patent (Section 9.2.3) on the real-time Linux approach. The idea is not really new, because Yaghmour found references from the early 90s. Philippe Gerum did most of the work in implementing the idea into a working piece of code. (Philippe also has complementary Free Software projects: Xenomai (http://freesoftware.fsf.org/projects/xenomai/) and CarbonKernel (http://freesoftware.fsf.org/projects/carbonkernel/), respectively aimed at real-time operating systems emulation and simulation.)

The following text is a copy from the `README` file of the Adeos code tarball: “*To share the hardware among the different OSes, Adeos implements an interrupt pipeline (ipipe). Every OS domain has an entry in the ipipe. Each interrupt that comes in the ipipe is passed on to every domain in the ipipe. Instead of disabling/enabling interrupts, each domain in the pipeline only needs to stall/unstall his pipeline stage. If an ipipe stage is stalled, then the interrupts do not progress in the ipipe until that stage has been unstalled. Each stage of the ipipe can, of course, decide to do a number of things with an interrupt. Among other things, it can decide that it’s the last recipient of the interrupt. In that case, the ipipe does not propagate the interrupt to the rest of the domains in the ipipe.*.”

10.2. eCos

(TODO: more details)

eCos (http://sources.redhat.com/ecos/) scheduler: fast and deterministic, deals with priority inversion, but not optimally; offers μITRON, POSIX and OSEK APIs and a non-standard API that shows its roots in the Cygnus company (“`cyg_scheduler_start()`” etc.). DSR: interrupts enabled but scheduling disabled. No kernel space/user space distinction. No development on same machine. Board support packages for a lot of processors, many of them embedded processors.

eCos has a quite turbulent history. RedHat acquires Cygnus in 1998 releasing their embedded operating systems efforts under the eCos name, but fires its eCos development team in June 2002. Development was taken over by eCos>entric (http://www.ecoscentric.com/). The license also changed over time, with the version 2.0 released under what is largely the GPL, with “guarantees” for compatibility with closed-source commercial components.

10.3. RT-EMS

The origins of RT-EMS (http://www.rtems.com/) lie with the Department of Defense in the USA, that wanted an Ada-based “Real-time Executive for Missile Systems.” This became the “Real-time Executive for Military Systems,” when they realised its relevance beyond missile control, and the C version later became the “Real-Time Executive for Multiprocessor Systems.” The Ada version keeps the “M” of “military.”

RT-EMS has a POSIX POSIX 1003.1b API (under construction); multitasking for homogeneous and heterogeneous multiprocessor systems; an event-driven, priority-based, preemptive scheduling; optional rate monotonic scheduling; intertask communication and synchronization; priority inheritance; responsive interrupt management; dynamic memory allocation; and it is compatible with the GNU tools.

(TODO: more details)

10.4. Jaluna

Jaluna (Jaluna (http://www.jaluna.com)) is an RTOS plus development environment released under a free software license in 2002. Jaluna is based on C5, the 5th generation of Sun Microsystems’ ChorusOS product.

10.5. Wonka + Oswald

Wonka (http://wonka.acunia.com) is a free software *Virtual Machine* for Java, with a real-time executive *OSwald*.

10.6. FIASCO and DROPS

FIASCO (http://os.inf.tu-dresden.de/fiasco) is a (for the time being academic) micro-kernel running on x86 CPUs. It is a pre-emptable real-time kernel supporting hard priorities. It uses non-blocking synchronization for its kernel objects, guarantees priority inheritance, and makes sure that runnable high-priority processes never block waiting for lower-priority processes. FIASCO is used as the kernel of the real-time operating system DROPS (http://os.inf.tu-dresden.de/drops/), thats want to bring *Quality of Service* to real-time operating systems.

10.7. Real-time micro-kernel

The Real-time micro-kernel (http://rtmk.sourceforge.net/) is inspired by the Mach micro-kernel, but is also meant for embedded systems.

10.8. KISS Realtime Kernel

The KISS Embedded Realtime Kernel (http://kiss.sourceforge.net/) is an academic project, intended for use in deeply embedded applications such as cell phones, cars, VCRs, consumer electronics, microwave ovens, toasters and ballistic intercontinental nuclear missiles. Being deterministic, small, readable and understandable, it is suitable for applications where deterministic response is primordial. The kernel also provides *resource tracking*: should an application terminate unexpectedly, all resources it had allocated are released.

II. RTOS implementation

This Part leaves the terrain of general concepts, and digs a bit deeper into implementation aspects of real-time operating systems. The RTAI operating system is taken as an illustration of a hard real-time operating system, and its implementation is explained in some more detail.

Chapter 11. RTAI: the implementation

(TODO: lots of things. Most sections are not decently structured, and their contents not decently checked. . .)

This Chapter describes the *implementation* of the three basic parts of RTAI: the hardware abstraction layer (RTHAL), the core of real-time task scheduling, and the “user space real time” LX/RT. The reader learns how RTAI can be a hard real-time kernel, while still allowing Linux to function “as usual” on the same hardware. The discussion doesn’t go into the deepest detail of the code however, but aims at offering the appropriate trade-off between detail and generality, to help the interested reader to quickly understand the (not extensively documented) RTAI source code, and to be able to place it in the wider context of (real-time) operating systems.

11.1. The RTAI source tree

The RTAI source code tree doesn’t reflect the subdivision into the major components presented in the previous Chapter: the Hardware Abstraction Layer, the Linux compatibility layer, the core functionality, LX/RT, and the extended functionality packages. So, finding where a particular feature is implemented can be time consuming. Part of the code, of course, contains hardware-dependent code, which contain the basis for the first three RTAI parts mentioned above. This code is concentrated in the following three directories (all directories given in this Chapter are with respect to the “root” directory of the RTAI source tree; or the Linux source tree, whenever applicable):

- `patches/`: this directory contains the Linux kernel patch, which is available for different Linux kernel versions and for different hardware architectures. The contents of the RTAI patch tend to change slightly from release to release, because of (i) a growing number of supported RTOS features that need low-level support; (ii) Linux itself evolving in the direction of offering a cleaner HAL, so eliminating the need for some parts of earlier patches; and (iii) code optimizations. It is necessary to apply the correct version of the patch to a *clean* Linux kernel of the corresponding version. And be aware that kernels that come with many Linux distributions have already been patched by the distributor for various reasons, so that patching it once more with the RTAI patch could fail.
- `include/asm-xyz/`, with `xyz` the identifier for a particular hardware; for example, `i386`, `arm`, or `ppc`. The header files in these directories also contain some code, often in the form of assembler in inlined function definitions.
- `arch/xyz/`, with `xyz` the identifier for a particular hardware. These directories, together with the above-mentioned header files, implement the hardware-dependent parts of RTAI’s functionality.

The state of the RTAI source tree at the time of writing is such that is doesn’t have clearly separated code trees for different stable and unstable versions. Hence, one sometimes finds different versions of a file in the same directory. For example, `include/asm-i386` contains several versions of `rtai.c`, with names such as: `rtai.c` (stable version), `allsoft.c` (experimental version), `rtai-22.c` (version for 2.2.x Linux kernels). The *configuration* scripts of RTAI choose the version that corresponds to your configuration selection, and copy them to the “official” filenames (which is `rtai.c` in the example above). Linux configuration, by the way, follows a similar approach. The RTAI patch also contains adaptation to the Linux configuration settings, such that existing Linux configuration tools can be used. For example, xconfig or menuconfig.

One of the most important files in the RTAI source tree is the *patch* to the Linux source tree. The patch modifies the Linux kernel, in order to place the "hooks" to which the RTAI functionality is attached. Such a patch file is in "diff" form (see the **diff** man page). That means that it lists only the *differences* between the original Linux source files and the adapted RTAI versions of these same files. This allows to keep the patch file small (far below 50 kilobytes) and to get a good and complete overview of the changes that RTAI applies. The diff file contains patches to different Linux files, each patch being of the following form (the markers at the end of lines are added for annotation purposes only):

```
diff -urN -X kernel-patches/dontdiff linux-2.4.18/Makefile ❶
linux-2.4.18-rthal5/Makefile                               ❶
--- linux-2.4.18/Makefile    Mon Feb 25 20:37:52 2002      ❷
+++ linux-2.4.18-rthal5/Makefile  Tue Feb 26 09:52:01 2002 ❷
@@ -1,7 +1,7 @@                                            ❸
 VERSION = 2                                               ❹
 PATCHLEVEL = 4                                            ❹
 SUBLEVEL = 18                                             ❹
-EXTRAVERSION =                                            ❺
+EXTRAVERSION = -rthal5                                    ❻
                                                           ❹
 KERNELRELEASE=$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)❹
                                                           ❹
```

❶ This line shows the **diff** command that has produced the patch.

❷ These are the files in two different directories whose **diff** is shown. One file is identified with minus signs, the other with plus signs.

❸ These are the line numbers, for both files, that the following part of the patch has changed.

❹ This is the "context" of the patch. The **diff** must always find *three consecutive lines* that have remained unchanged before and after the patched lines.

❺ This is the first part of the actual patch: the lines marked with "-" represent the code of the file identified previously with the minus signs.

❻ This is the second part of the patch: the lines marked with "+" represent the code of the file identified previously with the plus signs.

What the simple patch above does is filling in the *`EXTRAVERSION`* parameter that Linux provides to identify different build versions of the same kernel. In this case, the *`-rthal5`* identifier is added.

Here is a small list of "peculiarities" that (Linux and RTAI) kernel programmers tend to use quite often, and that could make reading kernel source code a bit tedious:

- *Magic numbers*: these are seemingly random numbers, that appear in many data structures. An example is found in the file `include/rtai_sched.h`:

```
#define RT_TASK_MAGIC 0x754d2774
```

This magic number is filled in in the *`RT_TASK`* data structure, in the function `rt_task_init_cpuid()` in the file `mupscheduler/rtai_sched.c`:

```
task->magic = RT_TASK_MAGIC
```

This data structure contains all information about an RTAI task. Since the kernel code is in C, and a lot of use is made of pointers to data structures, the magic numbers are used to check whether a pointer is pointing to the right data structure: if that is indeed the case, the magic number must be found at a prescribed place. In the *`RT_TASK`* example above, this check is performed many time in the scheduler code, as follows:

```
if (task->magic != RT_TASK_MAGIC) { return -EINVAL; }
```

where the error parameter *`EINVAL`* encodes an invalid situation.

- *`do {...} while(0);`*. This kind of construct appears quite often, especially in macro definitions in header files. At first sight, this seems a complicated procedure to execute the code between the braces just once, but in the context of macros it has a useful side-effect: using this *`while`* construct guarantees that compilers will not optimize anything away inside the construct, and they consider the whole construct as one single programming primitive (i.e., macro parameter), instead of the several individual statements that occur inside of the *`while`* scope. (See the kernelnewbies FAQ (http://kernelnewbies.org/faq/index.php3#dowhile) for more details.) One example is found in the RTAI patch (`patches/patch-2.4.18-rthal5g`):

```
-#define prepare_to_switch()    do { } while(0)
+#define prepare_to_switch() do {           \
+       if (rthal.lxrt_global_cli) {        \
+               rthal.lxrt_global_cli();    \
+       }                                   \
+} while(0)
```

- *`call *SYMBOL_NAME(rthal + 8)`*: these assembly language constructs are used to call a function at byte offset "8" in the *`rthal`* data struture. This is the data structure used for the hardware abstraction layer, Section 11.2. This complicated way to call a function allows to call different functions according to what is filled in in that data structure. RTAI uses it to replace Linux function calls with its own function calls. The patch files contain a couple of examples.

11.2. Hardware abstraction layer

The *RTHAL* (Real-Time Hardware Abstraction Layer), is, not surprisingly, *very* platform-dependent. Its code typically contains lots of assembler code that builds the low-level *infrastructure*, not only for the HAL, but also for the Linux compatibility layer (Section 11.3), the core (Section 11.4) and for LX/RT (Section 11.5). A large part of that code comes from RTAI's *patch*. The main patch fragments (as far as the HAL is concerned) are for the `arch/xyz/kernel/irq.c` and `include/asm-xyz/system.h` files in the Linux source tree. (Replace "`xyz`" with a suported architecture, such as arm, i386 or ppc.) The patch adds the *`rthal`* data structure to the `include/asm-xyz/system.h` file of the Linux source, and changes the interrupt handling and management functions that Linux uses. This *`rthal`* is the central data structure of RTAI's HAL: it collects the variables and function calls that Linux uses for interrupts (vector, flags, CPU affinity, i.e., the *hardware* abstraction), and task switching (which is the basis for the RTAI core, Section 11.4). In RTAI 24.1.9, the `rthal` data structure looks as follows:

```
struct rt_hal rthal = {
  void *ret_from_intr;                                       ❶
  void *__switch_to;                                         ❷
  struct desc_struct *idt_table;                             ❸
  void (*disint)(void);                                      ❹
  void (*enint)(void);                                       ❹
  unsigned int (*getflags)(void);                            ❹
  void (*setflags)(unsigned int flags);                      ❹
  unsigned int (*getflags_and_cli)(void);                    ❹
  void *irq_desc;                                            ❹
  int *irq_vector;                                           ❹
  unsigned long *irq_affinity;                               ❺
  void (*smp_invalidate_interrupt)(void);                    ❺
  void (*ack_8259_irq)(unsigned int);                        ❻
  int *idle_weight;                                          ❼
  void (*lxrt_global_cli)(void);                             ❼
  void (*switch_mem)(struct task_struct *, struct task_struct *, int);❼
  struct task_struct **init_tasks;                           ❼
  unsigned int *apicmap;                                     ❽
};
```

❶ Pointer to the “return from interrupt” call. By adapting this call, it’s not Linux but RTAI that decides what will be done next, after an interrupt routine has finished. (TODO: This pointer seems not to be changed any more during RTAI’s lifetime; is it still needed in the RTHAL?)

❷ Pointer to the function that does a task switch. Again, it should be RTAI that controls which task to switch to.

❸ Pointer to the *Interrupt Description Table (IDT)*, the data structure that holds the *status* of how interrupts behave: what is the interrupt service routine attached to an interrupt, what interrupts are enabled, and what are their priority and status.

❹ These lines contain the pointers to the fundamental interrupt *management functions* (disable and enable interrupts, with or without saving of the interrupt status flags), and data structures (interrupt descriptor (which IRQ to handle on which CPU). These pointers are filled in when RTAI is enabled. This happens in the function `__rtai_mount`, implemented in `arch/xyz/rtai.c`.

❺ These are only needed in an SMP system. The IRQ affinity remebers which interrupt numbers are possibly reserved to what CPU number; the data is filled in in `arch/xyz/rtai.c`. The `smp_invalidate_interrupt()` function is defined in Linux: `arch/xyz/kernel/smp.c`: a CPU in a multi-CPU system can raise a “request for TLB invalidate” interrupt to signal when a page in memory has been changed, such that others can take appropriate action to update their caches. RTAI can catch this interrupt, and decide when to give it to Linux.

❻ This is the function with which to acknowledge the interrupts from the timer. (The name is too much bound to the traditional 8259 timer chips; many others are in use nowadays.)

❼ This is used in LX/RT scheduling; see `lxrt/lxrt.c`. The `init_tasks()` function is defined in Linux: `arch/xyz/kernel/smp.c`.

❽ Points to a Linux-defined vector (in `arch/xyz/kernel/smpboot.c`) called "physical_apicid_2_cpu", which is filled at boot time and maps the physical APIC (Section 3.2) interrupt controller identifiers to logical CPU identifiers.

The form of the entries in the *Interrupt Descriptor Table* data structure is defined in Linux (`linux/irq.h`):

```
typedef struct {
   unsigned int status;        // IRQ status
   hw_irq_controller *handler; // functions to manage hardware interrupts
                                  // (see below)
   struct irqaction *action;   // IRQ action list
                                  // (see below)
   unsigned int depth;         // nested irq disables
   spinlock_t lock;            // lock used to access handler and
                               // action lists
} ____cacheline_aligned irq_desc_t;
```

This pointer to *`irq_desc_t`* is filled in in `arch/xyz/rtai.c`. The file `linux/interrupt.h` defines the *`irqaction`* field, that contains all information about a specific interrupt handler:

```
struct irqaction {
      // function to execute:
   void (*handler)(int, void *, struct pt_regs *);
   unsigned long flags;    // saved flags at moment of interrupt
   unsigned long mask;     // interrupt mask
   const char *name;       // name of handler
   void *dev_id;           // identifier of interrupting device
   struct irqaction *next; // pointer to next handler structure
};
```

And the file `linux/irq.h` defines the *`hw_irq_controller`* data structure:

```
struct hw_interrupt_type {
   const char * typename;
   unsigned int (*startup)(unsigned int irq);
   void (*shutdown)(unsigned int irq);
   void (*enable)(unsigned int irq);
   void (*disable)(unsigned int irq);
   void (*ack)(unsigned int irq);
   void (*end)(unsigned int irq);
   void (*set_affinity)(unsigned int irq, unsigned long mask);
};

typedef struct hw_interrupt_type  hw_irq_controller;
```

This data structure contains pointers to the functions needed to manage the hardware interrupts: how to enable and disable an interrupt, how to acknowledge an interrupts, how to attach an interrupt to a set of CPUs ("affinity"), etc.

The core of the HAL patch works as follows:

1. RTAI replaces Linux functions that work with the interrupt hardware with *pointers* to functions.

2. RTAI introduces the *`rthal`* data structure to store all these pointers.
3. RTAI can now switch these pointers to pointers to its own functions whenever it wants.

An example of this approach to replace original Linux functions with pointer entries from the *`rthal`* data structure can be seen in the patch to the `system.h` file:

```
#define __cli()               (rthal.disint())
#define __sti()               (rthal.enint())
#define __save_flags(x)       ((x) = rthal.getflags())
#define __restore_flags(x)    (rthal.setflags(x))
```

Here, the disable and enable interrupt functions are replaced, as well as the functions that save and restore the interrupt status flags. The patch, of course, also introduces "hard" versions of these functions, so that RTAI can work on the real hardware, while Linux works with the "soft" versions (i.e., the interrupts for these soft versions come from the RTAI software, and not from the hardware). For example, the `hard_cli()` and `hard_sti()` functions (that the patch adds to the Linux's `include/asm-xyz/system.h` file) get the functionality of the original `__cli()` and `__sti()` of Linux. This is again done in the patch file:

```
+#define hard_sti() __asm__ __volatile__ ("sti": : :"memory")
+#define hard_cli() __asm__ __volatile__ ("cli": : :"memory")
```

The original `__cli()` and `__sti()` of Linux are replaced by soft versions, as seen in the code example above.

Another (assembly code) example of the procedure to let Linux work with "intercepted" function calls, is the following patch fragment for the `arch/i386/kernel/entry.S` file:

```
 ENTRY(ret_from_fork)
+       sti
        pushl %ebx
        call SYMBOL_NAME(schedule_tail)
        addl $4, %esp
@@ -202,17 +203,20 @@
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)   # save the return value
 ENTRY(ret_from_sys_call)
-       cli            # need_resched and signals atomic test
+       call *SYMBOL_NAME(rthal + 12)                    ❶
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
+       sti                                              ❷
+       call *SYMBOL_NAME(rthal + 16)                    ❷
 restore_all:
        RESTORE_ALL

        ALIGN
 signal_return:
-       sti     # we can get here from an interrupt handler
```

```
+       sti   # we can get here from an interrupt handler  ❷
+       call *SYMBOL_NAME(rthal + 16)                      ❷
        testl $(VM_MASK),EFLAGS(%esp)
        movl %esp,%eax
        jne v86_signal_return
```

❶ The original `cli` (to disable interrupts) is replaced by a call to the function that sits on offset "12" in the *`rthal`* data structure. With four bytes per pointer, this corresponds to the fourth line of that data structure, i.e., the place where RTAI fills in its own version of the disable interrupt call.

❷ Similarly, but now for the function at offset "16", i.e., the enable interrupt function of RTAI.

All the adapted function definitions are finally filled in in the *`rthal`* data structure in the file `arch/xyz/kernel/irq.c`:

```
struct rt_hal rthal = {
   &ret_from_intr,
   __switch_to,
   idt_table,
   linux_cli,
   linux_sti,
   linux_save_flags,
   linux_restore_flags,
   linux_save_flags_and_cli,
   irq_desc,
   irq_vector,
   irq_affinity,
   smp_invalidate_interrupt,
   ack_8259_irq,
   &idle_weight,
   0,   // lxrt_global_cli
   switch_mem,
   init_tasks,
   physical_apicid_2_cpu
};
```

That is, they get pointers to their original Linux functions, or to patched functions that have the original Linux behaviour. The reason is, of course, that, at boot time, the system should behave as normal Linux. (Some of the entries in the *`rthal`* data structure have not been discussed yet, because they do not really belong to the *hardware* abstraction, but are meant to support the core functionality of RTAI, Section 11.4.) So, at boot time, Linux runs as if nothing has happened, except for a small loss in performance, due to the extra level of indirection introduced by replacing function calls by *pointers to function calls* in the *`rthal`* structure. The user can activate the RTAI functionality at any later time, via a loadable module that executes the `rt_mount_rtai()` (file `arch/xyz/rtai.c`). This switches the pointers to functions in the *`rthal`* data structure from their Linux version to their RTAI version. From that moment on, Linux is under control of the RTAI kernel, because Linux works with what it thinks is the "real" hardware through the replacement functions that RTAI has installed. But these functions give a *virtual* hardware to Linux, while RTAI manages the real hardware. For example, RTAI queues interrupts for Linux *in software* until Linux gets a chance to run again; at that moment, the pending interrupts seem to come from the hardware, as far as the Linux side is concerned.

In principle, the HAL could be used for other purposes than serving as a stub for the RTAI core. That is, another kind of operating system could be implemented on top of the RTHAL. But also the opposite could be done, i.e., implementing the same RTAI core on top of another low-level stub. This is what is being done in the ongoing integration of RTAI and Adeos (Section 10.1). This effort, however, experiences some problems because RTAI currently doesn't make a clean distinction between what is needed for a real *hardware abstraction* on the one hand, and what is needed for *replacing Linux* on the other hand. So, it is not straightforward to get the RTHAL alone, without any mention of the RTAI core or the Linux compatibility structures. For example, the patch and the `include/arch/xyz/rtai.c` file mixe both parts.

(TODO: explain implementation of all RTAI spinlocks and interrupt disabling/enabling functions; and `dispatch_trap` in `rtai.c` (checks whether traps come from NMI, Linux, debugger, of RTAI.); what do SRQs do? srqisr(),rt_request_srq() in rtai.c? use 0xFE as IRQ, *`#define RTAI_SYS_VECTOR 0xFE`* in `include/asm-i386/rtai_srq.h`? rtai_open_srq(); implementation of barrier `bits/rtai_bits.c`)

11.3. Linux compatibility layer

RTAI is developed to cooperate closely with Linux, and to let Linux take care of all non hard real-time tasks, such as networking, file IO, user interfacing, etc. But the cooperation with Linux is a *one-way* endeavour: Linux development doesn't (want to) care about how it could facilitate development of an RTOS below it. And its data structures are not fully appropriate for real-time. So, RTAI must place hooks in the Linux code, for the following things:

- Task data structures.
- Timing.
- LX/RT (Section 11.5): this requires interaction with Linux scheduling.

A first part of the Linux compatibility interface. consists of data structures in which RTAI stores the state in which it finds the running Linux kernel at the moment that it (i.e., RTAI) becomes active (`arch/xyz/rtai.c` and `include/asm-xyz/rtai.h`):

```
static struct rt_hal linux_rthal;
static struct desc_struct linux_idt_table[256];
static void (*linux_isr[256])(void);
static struct hw_interrupt_type *linux_irq_desc_handler[NR_GLOBAL_IRQS];
```

This state is restored when the RTAI module is unloaded. The Linux state is stored, and RTAI functionality is loaded, in the `init_module()` of `rtai.c`. This file, and its `include/rtai-xy.h` header file (with `xy` the RTAI version), further implement the basic function calls of a hard real-time kernel (Section 11.4). Note that *global* locks (similar to the *Big Kernel Lock*, Section 1.2, are available in RTAI. These locks, however, cannot be taken by user space processes, because the global locks in Linux have been virtualised.

The `rthal` data structure in the RTAI patch contains not only *hardware-related* fields (everything concerning interrupts), but also some *software-related* entries, such as task switching functions that have to work together with Linux. For example, the patch extends the `task_struct` data structure in `include/linux/sched.h` of the Linux source with

```
void *this_rt_task[2];
```

to accomodate a real-time tasks queue. The two pointers to real-time tasks are initialized to 0:

```
this_rt_task:      {0,0}
```

because at Linux boot, no real-time task exist yet. The other *Linux-compatibility* entries in the `rthal` data structure are:

```
struct rt_hal rthal = {
  void *ret_from_intr;
  void *__switch_to;
...
  int *idle_weight;
...
  void (*switch_mem)(struct task_struct *, struct task_struct *, int);
  struct task_struct **init_tasks;
...
};
```

The patch adds code to the source of the `linux/kernel/exit.c` file in the Linux source, to execute a *callback* to RTAI at the moment that a real-time task is stopped. In `linux/kernel/sched.c` of the Linux source, the scheduler is extended to work also with the LX/RT tasks (Section 11.5).

11.4. RTOS core

The RTOS core relies on the RTHAL and Linux compatibility "tricks" of the previous sections, to build a hard real-time kernel on top of the interrupt system of the hardware, and integrated with the task management of Linux. Chapter 8 gives more details about *what* functionality is offered; this section deals with *how* RTAI implements this functionality. The core's functionality consists of: task management and scheduling, interrupts and traps, synchronization and data exchange, and memory management.

The code of the RTAI core resides in the `include/asm-generic/rtai.h`, `include/asm-xyz/rtai.h` and `include/arch/xyz/rtai.c` files. The central data structure is the one that stores the global status of the RTAI core:

```
struct global_rt_status {
  volatile unsigned int pending_irqs;
  volatile unsigned int activ_irqs;
  volatile unsigned int pending_srqs;
  volatile unsigned int activ_srqs;
  volatile unsigned int cpu_in_sti;
  volatile unsigned int used_by_linux;
  volatile unsigned int locked_cpus;
  volatile unsigned int hard_nesting;
  volatile unsigned int hard_lock_all_service;
  spinlock_t hard_lock;
  spinlock_t data_lock;
};
```

11.4.1. Task management and scheduling

Task switching happens through the `switch_to` function in the `rthal` data structure; this can be seen in the

```
"jmp *"SYMBOL_NAME_STR(rthal + 4)"\n"
```

The function on offset “4” in `rthal` is indeed `__switch_to`. Also in this file is the trap handling; the relevant part in this assembler code is where the appropriate handler is called:

```
"call "SYMBOL_NAME_STR(lxrt_handler)"
```

This handler is filled in in (TODO). At the end of this assembler code, the “return from interrupt” is performed, again by calling the corresponding functions on the `rthal` data structure:

```
"1:call *" SYMBOL_NAME_STR(rthal + 16) "\n\t"
"jmp *" SYMBOL_NAME_STR(rthal)
```

Time management is very important for a real-time operation system, so RTAI has a bunch of functionality in its core to work with the time hardware. The low-level functions can be found in `arch/xyz/rtai.c`; for example:

```
int rt_request_timer(
  void (*handler)(void),
  unsigned int tick,
  int apic)

void rt_free_timer(void)

void rt_request_timer_cpuid(
  void (*handler)(void),
  unsigned int tick,
  int cpuid)

void rt_request_apic_timers(
  void (*handler)(void),
  struct apic_timer_setup_data *apic_timer_data)

void rt_free_apic_timers(void)
void setup_periodic_apic(
  unsigned int count,
  unsigned int vector)

void setup_oneshot_apic(
  unsigned int count,
  unsigned int vector)

...
```

11.4.2. Interrupts and traps

The “*encoded trap* ” technique consists of two parts:

1. Allowing a user space task to execute a kernel function.
2. Incorporating a user space task into the real-time scheduling. This requires an adaptation of the standard Linux task bookkeeping.

This first functionality is implemented via the use of a *trap* (Section 3.3). The trap allows the user space task to launch a kernel space function (the “trap handler”). The user space task *encodes* the desired real-time service in a set of two integers that it puts on the trap handler stack; it can, in addition, also pass some *arguments* to the trap handler. `dispatch_trap()` in `arch/xyz/rtai.c` does the trap handling.

11.4.3. IPC

locks, etc.

11.4.4. Memory management

`mlockall( MCL_CURRENT | MCL_FUTURE)`: POSIX function (coming from the Linux source tree: `linux/include/asm-xyz/mman.h`) that locks all pages of the calling task in memory; the parameters are macros that indicate that all current pages must be locked, but also all pages that the task will ask in the future.

11.5. LX/RT

LX/RT stands for “Linux/real-time”, i.e., it offers *soft* and *hard* real-time functionality to Linux user space tasks. This functionality is only slightly limited with respect to what can be achieved in kernel space. The ultimate goal is a fully “*symmetric API*”, i.e., to offer the same real-time API to user space tasks as what is available to RTAI kernel tasks. A symmetric API, available in user space, reduces the threshold for new users to start using real-time in their applications, but it also allows for easier debugging when writing new applications. The bad news is that it makes understanding the RTAI code a bit more difficult, because similar function calls get different implementations, depending on their usage in kernel space or in user space. This symmetry, obviously, can never be absolute and only works from kernel space to user space, and not the other way around: it is not possible to bring an *arbitrary* user space function to the kernel, because it would use functions that are not available in the kernel. Also, the user space task that one wants to execute in hard real-time via LX/RT should satisfy all constraints of hard real-time: no undeterministic or blocking calls, etc.

The LX/RT idea is quite old, actually, and has gone through various stages of evolution. The *first generation* used the idea to let a user space task run a companion task in the kernel, i.e., the so-called “buddy” in RTAI language. This companion task executes kernel space functions on behalf of the user space task. Technically speaking, this is realized by passing an identifier of the required function to a *trap handler*, which then executes the function call that corresponds to this identifier (Section 11.4.2); there is another “kernel space/user space” switch to return.

The *second generation* design of LX/RT (appropriately called *new LX/RT*) needs only one switch, doesn’t use a “buddy” anymore, and integrates maximally with existing Linux task scheduling. This means that Linux is not any more the “idle” task of the RTOS, but Linux itself has been extended with full pre-emption and real-time scheduling (for tasks that obey certain restrictions).

This clear distinction between first and second generation has only occurred *after* the facts: there have been several prototypes in various releases of RTAI, with names such as “LX/RT extended”, “ALLSOFT”, or “USP”. This has led to some confusion, but in the future only the two above-mentioned approaches will be supported.

From a user’s perspective, the *difference* between the soft and hard versions of LX/RT is that the hard version disables software interrupts when the LX/RT task runs.

11.5.1. LX/RT for the user

First, make the LX/RT functionality available by loading the LX/RT module, so that your tasks can use it. A typical LX/RT user task looks like this:

```
... TODO ...                                              ❶
```

❶

A maximum of `MAX_SRQ` Linux tasks can be made into hard real-time LX/RT tasks. (This constant is set to 128 in RTAI 24.1.9, in the file `include/rtai_lxrt.h`.) The user space task can also *register a name* for itself, consisting of at most six characters. This naming allows a LX/RT task to call all LX/RT functions via their “named” version; for example, `rt_named_task_init()`. The task name can also be used by other tasks than the one that gave the name, for example to send messages to each other.

A user space task is made into an LX/RT user space task by using only a couple of LX/RT calls. The task calls `rt_make_hard_real_time()` (in `include/rtai_lxrt.h`) at the moment it wants to switch to real-time, and `rt_make_soft_real_time()` to switch back. (Commenting out these functions is an easy way to allow user space debugging.) LX/RT also provides function calls to detect whether the calling task is currently running in hard real-time (`rt_is_linux()`, in `arch/xyz/rtai.c`) or in LX/RT: (`rt_is_lxrt()`), and whether it wants to use the floating point registers of the CPU.

(TODO: name registry.)

11.5.2. LX/RT implementation

(TODO: sketch the dependencies between files in LX/RT; document the encoding; stuff already done in `Documentation/README.EXTENDING_LXRT`.)

The `rt_task_init()`, implemented in the file `include/rtai_lxrt.h`. Its arguments are:

```
LX_TASK *rt_task_init(
   unsigned int tasknum,  // number of the task
   int prio,              // desired priority level
   int stack_size,        // allocated stack size
   int max_msg_size)      // max size of inter-task message buffer
```

This function call eventually ends up in the `__task_init()` in `lxrt/lxrt.c`, which initializes all parameters of the task and allocates the memory required for them. These are not only the parameters that the application programmer sets, but also the parameters needed behind the screens: the memory to communicate data to the trap handler, and the task data structure.

The real heavy part of LX/RT lies in `__lxrt_resume()` in `lxrt/lxrt.c`. This function takes care of the seemless integration with Linux task management. One of the calls it makes is to `__emuser_trxl()` in `XYZscheduler/rtai_sched.c` (where `XYZ` stands for “`up`” (uni-processor), “`mup`” (multiple uni-processors), or “`smp`” (symmetric multi-processor):

```
static inline void __emuser_trxl(RT_TASK *new_task)
{

  if ((new_task->state |= READY) == READY) {              ❶
    enq_ready_task(new_task);                             ❷
    rt_switch_to_real_time(0);                            ❸
    save_cr0_and_clts(linux_cr0);                         ❹
    rt_switch_to(new_task);                               ❺
    if (rt_current->signal) { (*rt_current->signal)(); }  ❻
  }
}
```

❶ ...

❷ ...

❸ ...

❹ ...

❺ ...

❻ ...

Two important functions in `lxrt/lxrt.c` are: `steal_from_linux()`, and `give_back_to_linux()`:

```
static void steal_from_linux(RT_TASK *rt_task)
{
   int cpuid;
   struct klist_t *klistp;
      ...
   rthal.lxrt_global_cli = linux_lxrt_global_cli;          ❶
   cpuid = ffnz((rt_task->lnxtsk)->cpus_allowed);          ❷
   klistp = klistbp[cpuid];                                ❸
   hard_cli();                                             ❹
   klistp->task[klistp->in] = rt_task;                     ❺
   klistp->in = (klistp->in + 1) & (MAX_SRQ - 1);          ❻
   hard_sti();                                             ❼
   current->state = TASK_LXRT_OWNED;                       ❽
   wake_up_process(kthreadb[cpuid]);                       ❾
   schedule();                                             (10)
   rt_task->is_hard = 1;                                   (11)
   HARD_STI();                                             (12)
   if (current->used_math) { restore_fpu(current); }       (13)
}
```

❶ ...

❷ ...

❸ ...

❹ ...

❺ ...

❻ ...

❼ ...

❽ ...

❾ ...

(10)...

(11)...

(12)...

(13)...

```
static void give_back_to_linux(RT_TASK *rt_task, int in_trap)
{
  int cpuid;
  struct klist_t *klistp;
   ...
  cpuid = ffnz((rt_task->lnxtsk)->cpus_allowed);
  hard_cli();
  if (in_trap) {                                             ❶
          rt_signal_linux_task((void *)0, 0, rt_task);       ❷
  } else {
          klistp = klistep[cpuid];
          klistp->task[klistp->in] = rt_task->lnxtsk;
          klistp->in = (klistp->in + 1) & (MAX_SRQ - 1);     ❸
          rt_pend_linux_srq(sysrq.srq);                      ❸
  }                                                          ❸
  rem_ready_task(rt_task);                                   ❸
  lxrt_schedule(cpuid);
  rt_task->is_hard = 0;                                      ❹
  hard_sti();
}                                                            ❺
```

❶ ...

❷ ...

❸ ...

❹ ...

❺ ...

LX/RT uses Linux kernel threads `kthread_b()` (ldquo;*kernel thread begin*”) and `kthread_e()` (ldquo;*kernel thread end*”), with its own `lxrt_schedule()` scheduling.

(TODO: what do `kthread_b()` and `kthread_e()` really do?)

RTAI has had different versions of LX/RT functionality. The later ones are more robust against a task crash in the user space Linux side. At that moment, Linux executes a `do_exit()` function, and the RTAI patch has added a pointer to a callback function in that function. The callback is used to free the resources that where registered by the real-time buddy. It also deletes the real-time buddy task, and unblocks any other task that may have engaged in blocking IPC (e.g., via a semaphore) with the real time task.

This section discusses the implementation of the LX/RT techniques, (at the time of writing, only fully supported on i386 hardware) as implemented in the following files in the RTAI source tree:

- `include/asm-i386/rtai_lxrt.h`

- `include/rtai_lxrt.h`
- `lxrt/lxrt.c`
- `ABCscheduler/rtai_sched.c`, where `ABC` stands for either `up` (*uni-processor*), or `mup` (*multiple uni-processors*), or `smp` (*symmetric multi-processors*).

This involvement in LX/RT of a file called "scheduler" is one of these unfortunate things that make RTAI a confusingly documented project… The reason it is needed in the discussion on LX/RT is that it contains the implementation of the *RTAI kernel space* function `rt_task_init()`, which starts up a kernel space "proxy" (or `buddy_fun()` as it is called in `lxrt/lxrt.c`) for each user space LX/RT task that calls the *user space* function with the same name, `rt_task_init()`, but with different arguments. This `buddy_fun()` function has, at first sight, a strange implementation:

```
struct fun_args { int a0; int a1; int a2; int a3; int a4; int a5; \
   int a6; int a7; int a8; int a9; long long (*fun)(int, ...); };

static void buddy_fun(RT_TASK *mytask)
{
  struct fun_args *arg = (void *)mytask->fun_args;
  while (1) {
    mytask->retval = arg->fun( \
      arg->a0, arg->a1, arg->a2,  arg->a3, arg->a4, \
      arg->a5, arg->a6, arg->a7, arg->a8, arg->a9);
    lxrt_suspend(mytask);
  }
}
```

So, *every* LX/RT task gets this same `buddy_fun()` as its buddy. But yet the result of executing it differs from task to task, and from activation to activation of the buddy, because the buddy executes the function (and the arguments) that it got in the *list of parameters* from the `lxrt_handler()` trap handler (Section 11.4.2). So, the `buddy_fun()` remains suspended until the user space task makes a real-time LX/RT call; at that moment, the LX/RT scheduler wakes up the buddy with `lxrt_resume()`, which executes the function that it got through its arguments from the trap handler, and then goes to sleep again.

LX/RT has 16 *expansion slots*, that application writers can use to plug-in their own functionality. That means, if you have a set of functions that run in hard real-time, by the LX/RT extension you can make them available to user space tasks. You need to do two things:

- *In kernel space.* done by acquiring (at compile time of your functions) a "key" from LX/RT, and making an array of function pointers. So, you functions can be recognized by the LX/RT infrastructure based on these two numbers: (i) the LX/RT key, and (ii) the index in the function array.
- *In user space.* Make user space functions with the same interface as the above-mentioned kernel space functions, by using the `rtai_lxrt()` function that takes care of the trap setup, discussed in a section above.

The symmetry between the use of the new functionality in kernel space and user space shows up your source files: their typical structure is as follows:

```
#ifdef MODULE
 ...
#ifdef __KERNEL__
```

```
        // kernel space functions
#else
        // user space functions
#endif

#endif
```

The above-mentioned functions such as `rt_task_init()` and `rt_make_hard_real_time()` are examples of the “*encoded trap* ” technique (Section 11.4.2) that is behind all of LX/RT. The argument passing needed in an encoded trap is performed in the short but somewhat enigmatic function `rtai_lxrt()` in `include/asm-xyz/rtai_lxrt_sup.h`. The “magic” is due to the argument encoding used in this function.

So, the clue of the LX/RT procedure is to make the user space task launch a trap handler that executes a real-time service for the user space task; and all this is done through just one single trap handler, by encoding the desired service. Hence, a special LX/RT version must be made for *all* RTAI functions that one wants to be available to user space tasks, and a unique code must be given to each function. The codes are given in `include/rtai_lxrt.h`, which also contains the LX/RT versions of the core RTAI calls; non-core functionality (FIFOs, serial communication, etc.) needs extra header files, such as, for example, `include/rtai_fifos_lxrt.h`. The function `lxrt_handler()` in the file `lxrt/lxrt.c` connects the encoded service requests with real executable calls in kernel space. The `lxrt_handler()` does not only work with *tasks* shared between user space and kernel space, but also with *semaphores* and *mailboxes*, via precisely the same technique: encoding what the desired action is, in the data given to the LX/RT trap.

`dispatch_trap()` in `arch/xyz/rtai.c` does the trap handling. If it is a trap for RTAI it is passed to the trap handler that RTAI has registered; this is done in the `init_module()` of `lxrt/lxrt.c`. `lxrt_trap_handler()` in `lxrt/lxrt.c`: catches 7 (floating point error?) and 14 (memory allocation error), and then proceeds to the basic job:

The data structure for the coded trap message from user space to kernel space is hardware dependent, e.g., for i386 it is defined in `include/asm-i386/rtai_lxrt.h`:

```
union rtai_lxrt_t { RTIME rt; int i[2]; void *v[2]; };
```

It is a *`union`* data structure, because its contents can have various meanings. The same file also shows that for this hardware platform, LX/RT chooses the trap *`int $0xFC`*; this trap number is defined as:

```
#define RTAI_LXRT_VECTOR  0xFC
```

but also occurs directly in the assembler code that defines the data structure for the trap:

```
static union rtai_lxrt_t _rtai_lxrt(int srq, void *arg)
{
  union rtai_lxrt_t retval;
  __asm__ __volatile__ ("int $0xFC"
  : "=A" (retval) : "a" (srq), "d" (arg));
  return retval;
}
```

Incorporating a user space task into the real-time scheduling is the second LX/RT funtionality. It is implemented by patching the normal task switching code of Linux. `include/asm-i386/rtai_lxrt.h` defines the hardware-dependent part. LX/RT works with a flag that signals the scheduler whether or not to take into account LX/RT tasks; this flag

```
volatile unsigned long lxrt_hrt_flags;
```

keeps track of whether a task is running in hard real-time or not. The flag is used, for example, in the `XYZ/scheduler/rtai_sched.c` file. That scheduler code also uses the *`my_switch_to`* function. Also `lxrt/lxrt.c` uses that function, in the `lxrt_schedule()`. That is a replacement for the Linux `schedule`. `lxrt_schedule()` is used in the kernel thread scheduler (??) `kthread_b()`, and in `give_back_to_linux()`

Scheduling in LX/RT: `lxrt_sigfun()` to `lxrt_schedule()` when getting back to Linux from the RTAI schedulers; `steal_from_linux()` to make a Linux process a user space hard real-time module; `give_back_to_linux()` to return a user space module to the Linux tasks.

(TODO: signals for LX/RT tasks.)

11.6. Making your own extensions to LX/RT

(TODO: LX/RT/Comedi as an example of extending LX/RT.)

11.7. Module implementations

This section explains the code in the `init_module()` functions in the various RTAI parts.

`init_module()` of `lxrt/lxrt.c` does the following: …

Chapter 12. C++ and real-time

Operating systems are most often written completely in one single language, and most often that language is C. There are (and will be) always small, hardware-dependent parts that use assembly language, for efficiency or feasibility reasons. But, at the other end of the language spectrum, also object-oriented languages are being used; sometimes in combination with C as the basic, low-level language, [Walmsley2000].

12.1. C and C++

Most operating systems are programmed in C: all commercial UNIX systems, Linux, Microsoft NT, QNX, etc. Writing operating systems in a hardware-independent way was exactly the reason why Bell Labs created the C language. It is not much more than an embellished assembly programming language, but it has become a de facto standard because of the success of the UNIX operating system. The *pointer* concept of C is one of its major advantages for writing device drivers and operating systems: it allows the programmer to place a variable of a program onto a specific hardware address; or to work with (doubly) linked lists of data structures, which is a very common need in the bookkeeping tasks of the OS; etc. Efficiency is another advantage of C: is doesn't have a "runtime" (such as C++ or Java) in which non-deterministic operations take place behind the screens (e.g., dynamic allocation of memory; garbage collection), and beyond the control of the programmer.

C does have a number of disadvantages too, of course. Modern programmers have learned to appreciate object-oriented programming, with its emphasis on keeping related data and functionality hidden inside of classes with well-defined interfaces. Although a programmer *can* practice the ideas of object-oriented programming in C, the language itself doesn't support it. And a large part of the C source files of free software projects prove that writing "spaghetti" programs is way too easy in C. . .

So, Ada (in the 1980s already) and C++ (from the late 1990s) have appeared on the radar screen of operating system programmers. Not Java, or Eiffel, or other object-oriented languages, because Ada and C++ allow to keep most of the C advantages (pointers, efficiency) needed in operating systems. Ada became in vogue because the US army wanted a reliable and "safe" programming language for all its real-time and embedded software. Ada is still mandatory for most aerospace systems (military as well as civilian). RT-EMS (Section 10.3) is a free software RTOS that came into being in this context.

An interesting evolution in portable and high-quality C++ code, that can (sometimes) be used in real-time systems, is the Boost project. (http://www.boost.org) The project offers interesting motivations (http://www.boost.org/libs/thread/doc/rationale.html) for their work on threads, locks, etc.

The three primary aspects of object oriented programming are

1. *Encapsulation.* The idea to encapsulate the implementation of a class is based on various motivations:

- To distinguish between the *interface* (or, “*specification*”) the class, and the *implementation* of the operations that can be called on the class.
- The need for *modularity*, in order to structure complex applications designed and implemented by a team of programmers.
- It offers a structure for protection and authorization.

2. *Inheritance.* The idea that one class can inherit properties from another class provides for a *natural classification* of classes, with a minimum of specifications. Natural means that the software specifications of a class correspond closely to the properties that we know from real-world objects and/or concepts. The inheritance relationship between classes makes one class the *parent* (or “base”, “superclass”, “ancestor”, etc.) of another class. Inheritance can be used as an *is-a-kind-of* (or *is-a*) relationship. Inheritance comes in two flavours: *interface inheritance,* and *implementation inheritance*.
3. *Polymorphism.* This is the idea that the same software object can behave in different ways, depending on various factors.

Not all object-oriented languages offer the full set of these concepts, or implement them in the same way. For example, C++ lacks direct support of the concept of a *interface,* such that interface inheritance is always implemented by “workarounds”. (TODO: how?)

None of the above-mentioned object-oriented aspects are supported in C, which leads to the following “problems”:

- *Lack of encapsulation.* If the programming language doesn’t impose or stimulate encapsulation, any effort at trying to separate specification from implementation that the original coder may have had, tends to be compromised very quickly, not only by other contributors, but also by the original coders themselves.
- *Lack of inheritance.* Because C programmers have never been taught and drilled to watch for commonalities between different software components, most of them apply code re-use by “copy-and-paste” of source code lines between components. But once a common piece of code appears in two different places, these two pieces begin to have their own evolution, and the bonus of having a common ancestor disappears, and the code becomes larger. Typically, newcomers to the project don’t know about the commonality insights their predecessors had, and have much more problems understanding the code, and will, hence, be less efficient and more error-prone in their contributions.
- *Lack of polymorphism.* This has led to the introduction of “states”, with Boolean operators as the simplest form of state: the “class” reacts differently to the same inputs when it is in a different state; or, alternatively, it accepts different inputs according to its state. This in itself is not the real issue, but in combination with the lack of interfaces and encapsulation, the internal states of objects are used by other software components, that begin to adapt their interactions with the object based on the knowledge of its state. This is a typical situation of *high coupling* between software components; Chapter 14 explains the pitfalls of this situation.

Of course, using the above-mentioned OO aspects is not *in itself* a sufficient condition for writing good quality software!

12.2. C++ in the Linux RTOSs

C++ has kept all aspects of C that are useful in the implementation of software that works in close interaction with the hardware: pointers to hardware addresses being a major features. However, C++ also has two parts whose execution is non-deterministic: (i) dynamic object creation and deletion, and (ii) the Run-Time Type Identification (RTTI) feature. The good news is that both parts are reasonably easy to avoid, because there is even compile-time support from the compilers to disable these non-deterministic parts. The bad news is that most of this functionality is used deeply behind the screens of object creation and deletion, and exception handling; and few programmers have been trained in spotting these points in their C++ code.

Both RTLinux and RTAI have growing support for C++ in real-time, but the majority of their programmers and code are not “C++-ready”. eCos, on the other hand, has been written completely in C++ from scratch, and hence all of its contributors must master the C++ basics.

RTAI (Section 8.10) allows to use C++ in kernel space. But this does *not* mean that one can use Linux kernel functions from C++. That will most likely cause problems when trying to include Linux kernel header files into RTAI C++ files, and similarly with RTAI header files. To get the functionality that is needed for the `rtai_cpp` classes, some wrapper file was “hacked”, to deal with just a very few problem headers. This file does not give full Linux kernel functionality to C++ programs, though. So, in order to use a function from Linux, one needs to wrap it in an `extern "C"` function.

Chapter 13. Cross compilation, debugging and tracing

This Chapter explains the basic principles behind developing code for another platform than the development platform, loading code to that platform, making it boot autonomously, and debugging it. Tracing of the execution of a running embedded or real-time system is another important tool to assess the behaviour of an application in its whole.

13.1. Cross development

(TODO: How? What hardware support needed?)

13.2. Debugging

(TODO: host + target, remote debugging, BDM, S-records for serial communication, `rt_printk()` (explain where its output ends up).

13.3. Linux Trace Toolkit

(Excerpt from the documentation of Linux Trace Toolkit.) The Linux operating system is a multiprogramming, multiuser system. This means that it is able to handle multiple running programs at once. On a uniprocessor system (a computer with only one microprocessor), this is achieved by sharing this precious resource among multiple tasks, letting each execute for a certain period of time and then switching to another. The selection and switching of processes is handled by the Linux kernel, which also is a program and therefore also needs some time from the processor. It is also responsible for fulfilling certain requests by the programs it manages, dealing with error conditions, etc. One could have the need to know exactly what these schedulingdecisions, process switches and various management tasks are, how they are handled, how long they take and to which process the processor is allocated. Spread out over a certain period of time, we call this an execution trace.

The Linux Trace Toolkit (http://www.opersys.com/LTT/), is a suite of tools designed to do just that: extract program execution details from the Linux or RTAI operating systems and interpret them. Specifically, it enables its user to extract processor utilization and allocation information for a certain period of time. It is then possible to perform various calculations on this data and dump this in a text file. the list of probed events can also be included in this. The integrated environment can also plot these results and perform specific searches.

Linux Trace Toolkit works by inserting tracing commands into the source code. This requires a “patch”, and the extra instructions slow down the execution a little bit.

(TODO: more details.)

III. Design

The first Parts of this text dealt with *functionality*; this Part is about *structure*.

The previous Chapters described real-time operating system concepts, their *general-purpose functionalities*, and some of their implementation aspects. The following Chapters give information, examples and hints about how to *design* applications on top of this raw RTOS functionality. Indeed, design is not about adding as many RTOS features in your application as you can, or about using the first API function you find that could be used to solve your current implementation problem. No, design is all about making the *logical structure* of your particular application as explicit and clear as possible, and on searching hard to reduce the number of RTOS features needed to implement that application logic. So, design is, by definition, always driven by *application-specific* criteria, and hence no “general purpose” real-time system design exists. Nevertheless, there *are* lots of generic design issues, that have received neat solutions that all application programmers should be familiar with. Examples treated in later Chapters of this Part are: loose coupling, components, application architectures, software patterns, and frameworks.

The observation from which to start a discussion on design, is that (mature) application domains have, over the years, developed a relatively fixed *structure* of cooperating tasks. But, typically, the *functionality* of some of the tasks changes more quickly: features are added, alternative implementations of functionality are tried, new hardware or communication protocols are supported, etc. Examples of such mature domains are: telecom, motion control of machines, setpoint control of processes, networking, or data bases. Except for the last domain, real-time and embedded aspects are very important. Hence, it’s the goal of this Part to describe *good designs* for the basis *structure* code in these application domains. The domain of *general feedback control and signal processing* is taken as an example. (And similarly worked-out examples for other domains are very welcome!) The good news is that most application domains have a lot of very similar basic needs. The Software Engineering community is working hard to capture these in so-called *Software Patterns*; the ones relevant to real-time and embedded systems are presented later in this Part.

Most of the material addressed in this Part is not unique to real-time or embedded systems. But real-time and embedded systems tend to be more affected by the “holy grail of efficiency”, which is one of the major causes of poorly structured and hence poorly maintainable code: programmers make “shortcuts” that mix *functional aspects* of the *application* with *structural* aspects of the application’s *architectural design*. And mixing function and structure makes implementations much more messy. We *do* pay attention to efficiency, however. And the focus remains on real-time systems, which means that *scalability* comes second: no-one expects a real-time application to be scalable to the same extent as other IT applications, such as web or database serving. But a well-designed project has a *structure* in which it is clear which parts are scalable and which are not.

Chapter 14. Design principles

The message of this Chapter is: a well-designed software project has a clear and documented *decoupling between structure and functionality*. Section 14.1 explains what “structure” and “functionality” mean, and Section 14.2 says how to separate them. Components (Section 14.4) are the modern software engineering approach to build software projects that use cleanly separated structure and functionality. The architecture of the system (Section 14.5) defines how the available components are to be connected together, and explains why that particular choice should be made.

This Chapter only talks about the *theory* of good software design; the following Chapters illustrate this theory with applications that are relevant to the scope of this document.

14.1. Structure and functionality

Functionality is the set of all algorithms needed to perform the purpose of the application; *structure* is the way in which the algorithms are distributed over tasks, which tasks have to exchange data, and how they have to be synchronized. In large software projects, the division between structure and functionality is important, because few of the contributors will be able to grasp the whole software project and predict the consequences of the code they add to the project. Therefore, the “senior” project people should define and code the project’s infrastructure, in which all contributors can add functionality while having to consider only a limited part of the project. This idea is already wide-spread, because large projects such as Mozilla, Gnome, the Linux kernel, Windows 2000, etc., all work (more or less) along these lines. However, a lot of the real-time developments (outside of specialized companies) are done in small, isolated groups, where often the majority of developers are not computer scientists but specialists in the application domain; and new students in the field are concentrating more on understanding the real-time primitives than on learning to design software applications. This often leads to “spaghetti code”, and abuse of the available real-time and IPC primitives.

These are some examples of large scale applications, with major real-time and embedded needs:

- *Communication networks*: examples of structure in this domain are the *OSI* 7-layer model, the principle of *name servers*, the *CORBA* specifications, etc. They all decouple the infrastructure of sending packages with data from the meaning of these packages in the context of an application. Telecom is one of the best examples of time-prove designs (and of the importance of *open standards* to make optimal use of these designs): everybody takes it for granted that telephone systems do not crash, but few people realise the magnitude of software components involved in the process.
- *Control systems*. Many devices and even whole plants are controlled by computers. These systems have a wide variety of *functionalities*: oil refinery, milling tools, medical apparatus, laser “guns” in discotheques, laser printers, etc. But they all have the same basic components, well-known and thoroughly studied in engineering sciences such as systems and control theory. The generic *structure* of all these applications is that of *feedback control*: one component generates a signal that represents the desired value(s) of one or more of the physical signals in the system; one component measures these physical signals; aenother component derives values of other signals that are not directly

measurable; one components steers the inputs to "actuators" that can change the value of (some of) the relevant physical signals. Chapter 16 gives a more detailed description.

The control application is used in the following Chapters to illustrate the theory of good design with practical examples.

14.2. Loose coupling

Only the simplest of applications can be programmed as one single task, with nothing else but straightforward communication with peripheral hardware. Most other projects need multiple components, and hence synchronization and data exchange between them. Many developers make the mistake of putting too much *coupling* between their components. That means that, in the implementation of one software component, they use knowledge about the implementation internals of other software components, or about the specific architecture (and operating system!) in which the components are used. Some typical examples are:

- A task in one component suspends and restarts a task in another component. This implicitly means that the first task knows (or rather, pretends to know!) when it is appropriate and safe to influence the activities of the other task.
- Component A uses a *finite state machine* to structure its internal working; component B bases its interaction with component A on deciding which state it wants A to go to. This means that the implementation of A cannot be changed without changing B also.
- Task X delays itself during 100 milliseconds, in order to allow task Y to be scheduled and get started. This means that the "proper" synchronization of X and Y depends on platform and configuration dependent timing; and this timing tends to change drastically when functionality is added, or hardware is updated.
- There is only one interrupt service routine in the application, and the programmer doesn't do the effort of splitting its implementation into a *real* ISR (that does nothing but the really essential processing) and a DSR (that takes care of the further processing of the interrupt, Section 3.4). Again, this is a software structure that is error prone when updating the system or migrating it to other hardware.
- Application programmer S raises the priority of a task, because it doesn't run "fast enough" within the current situation. She also switches to *priority inheritance* (Section 4.8) for some of the system's critical section. Her colleague H adds his part of the system, and also feels the need to raise the priorities of "his" tasks and critical sections. This phenomenon reflects an implicit use of knowledge about the operating system, and, in practice, often leads to a race that eventually ends in most tasks running at the highest priorities, which, of course, reduces the usefulness of priority-based scheduling.

(TODO: make this list of bad examples as exhaustive as possible.)

The solution to these coupling problems is, of course, quite simple: *avoid every form of coupling*. Or rather, strive for *loose coupling*, because complete decoupling is only possible for tasks that have nothing to do with each other. This loose coupling advice, however, is difficult to translate into concrete guidelines. It's one of these things that make good (software) engineering stand out from the rest, and makes program design into an "art". Understanding what exactly causes the problems in the list of examples given above, is already a good beginning; as is understanding the relevant software patterns in

Chapter 15. Looking at your application in an object-oriented way also helps a lot: the fundamental reason behind the successful use of objects is exactly their ability to clearly *decouple* structure (“class hierarchies”) and functionality (“object methods and data”). And designing a software system in an object-oriented way is independent of the language(s) used in the implementation of the system. A good design can also be implemented in C (if the programmers are disciplined enough). Anyway, there is more to a software system than describing which objects it should use…

14.3. Mediator

A *mediator* is a major design principle to introduce *loose coupling* (Section 14.2) into the interaction between two or more components. “Mediator” means “object in the middle” between two interacting objects. And “interaction” means, at least, both *synchronization* and *data exchange*, i.e., IPC. The mediator takes care of the decoupling in multiple ways:

- *Service name serving*: the components need not know each other’s identity, but just have to know the name of the mediator they have to interact with, in order to get the service they are looking for in the interaction.
- *Data handling*: a general interaction involves the exchange of data between the interacting components. This data resides in the object or component that implements the mediator, and the interacting components access the data through methods in the mediator’s interface. Only in this way, the mediator can guarantee data consistency. Hence, the implementation of the mediator software pattern will itself be based one some sort of *monitor* (or “*protected object*”) software pattern (Section 15.2).
- *Synchronization*. Instead of distributing over all the interacting components the information about how to synchronize their mutual interaction, it’s only the mediator that has to know about the synchronization needs in the whole system of interacting components. And it’s only with the mediator that each component interacts. This means that a “graph-like” interaction structure is replaced by a much simpler “star-like” interaction, with the mediator in the centre.

In summary, the mediator pattern is one particular *policy* to use the *mechanism* offered by the monitor pattern. In addition, the monitor patterns makes use of several (purely) *synchronization* mediators. This is not a contradiction, or a case of circular reasoning: mediator and monitor patterns exist in different levels of complexity. It’s the more complex form of the one that makes use of the simpler form(s) of the other.

The following Chapters introduce various examples of mediators. With a small stretch of the imagination, all synchronization (Chapter 4) and data exchange (Chapter 5) IPC primitives can be considered to be mediators; however, in their implementation this insight has rarely been taken into account.

14.4. Components

A *component* is a unit of software with a clearly specified *interface*, that describes its functionality. This functional interface makes a software component into an independently reusable piece of code, i.e., something that can be used as a “service”. It’s this independent server property that distinguishes a component from an object class. Here is a list of things that a component *can* (but need not) do: run several tasks and IPC channels internally; interrogate other components to find and use their interfaces; be interrogated itself; and generate and handle events. Hence, a component is deliverable by third parties on the basis of its interface specification only. In principle, it can even be delivered as a binary. A component should not only document the interface it offers to the world, but also the interfaces it requires from other components in order to do its job.

The difference between (or rather, the complementarity of) the concept of *components* and the concept of *object-oriented programming* is that object-oriented programming talks only about *static* structure, and not about concurrency and system semantics. Even the most compilers for object-oriented languages cannot impose that, for example, the `initialize()` method of an object must be called *before* its `run()` method. Synchronization constraints in a system also don't enter the scope of object-oriented programming. For example, `method x()` must be executed every millisecond, but only when `method y()` has run successfully.

Different implementations of the same interface can focus on different optimizations: efficiency of execution or of memory usage, proven absence of deadlocks, etc. In a typical application, components of different types are needed, and must be integrated. A component-based design facilitates the distribution of the implementation work over several contributors, it improves portability, and facilitates the integration with, and re-use by, other projects.

So much for the theory... This theory seems to work quite well in the context of "normal" applications (business transactions processing, web services, etc.), but in the context of real-time and embedded systems, the classical component interface lacks two important specifications:

- *Timing and synchronization needs.* It's next to impossible to guarantee in a component interface what the exact behaviour of the component will be if it is integrated into other people's real-time application.
- *Memory needs.* The memory footprint of the component binary in itself is not sufficient information for its safe integration into an embedded system: the component could use much more heap and stack space during its operation than expected.
- *Composition of components.* "Glueing" two components together doesn't, in general, result in a new component. That is, the composite "component" doesn't offer a new interface consisting of an ordered sequence of completely specified interactions and properties. It is for this reason that the development of concurrent programs is an error-prone process. Especially in real-time systems.

The conclusion is *not* that the idea of components should be abandoned for real-time and embedded system; only that the use of binary delivered third-party components is cumbersome. The advantages of specifying clear interfaces of parts of a system remains a useful design guideline, also for real-time and embedded application programmers, and even within a project that has no externally delivered components. A real-time system should have a good description of which components it offers (i.e., describing their *functionality* in a documented interface), and how their *internal* subcomponents are interconnected (i.e., the *structure* of the system). This mostly boils down to deciding which tasks to use, and what their synchronization and data exchange needs are.

The pessimistic view is that this specification of the internal structure violates the previously introduced guideline for loose coupling, and that this violation is often unavoidable. The optimistic view is that this internal structure is a *natural part* of a good software component: as said before, and as explained in more detail in Chapter 15, a mature software solution for a particular problem has a natural structure which has proven to be the best. Hence, revealing this structure should not be seen as a negative compromise.

14.5. Architecture

The architecture of a software system is all about choosing its *structure*: it is a *specific connection* between the software components in the system. That is, an architecture makes a specific choice of how data and signals must travel between components. And, in general, other structures may be possible.

The best-known example of a system architecture is probably the *hierarchical architecture*: data and signals flow from layer to layer in the hierarchy. This architecture has the advantage of being very transparent and hence understandable, but the disadvantage of being inflexible. Many projects start this way, but then try to “patch” the architecture later on, when trying to work around the inflexibility. An example of an often encountered inflexibility is that the highest level in the hierarchy is, strictly speaking, not allowed to investigate the status of the components in the lowest layer directly, and must pass through all intermediate levels.

So, in the design phase one should try to postpone decisions about the architecture as long as possible, and to design the components in such a way that their functioning does not depend on a particular choice of architecture. It is indeed almost certain that the requirements of the software project will change during its lifetime, and architectures are often the most difficult to adapt aspect of a software system. Especially so when the original developers did not provide an explicit description and motivation for the system’s architecture.

Chapter 15. Patterns and Frameworks

This Chapter defines what *software patterns* and *software frameworks* are. It describes some common patterns that are relevant to real-time systems: monitors, events and state machines, and mediators such as “producer-consumer” and “execution engine”. The motion control *framework* is developed in more detail in Chapter 16, with the presented software patterns as “medium-level” components, and the RTOS primitives of Part I in *Real-Time and Embedded Guide* as “low-level” programming primitives.

15.1. Definitions

A *Software Pattern* ([gof94], [posa96]) is a *proven, non-obvious, and constructive solution to a common problem in a well-defined context*. This solution is the result of many years of experience dealing with the often delicate interactions and trade-offs (“forces”) that each tend to drive the solution into different directions. A Pattern describes the interaction between a group of components, hence it is a higher-level abstraction than classes or objects. It’s also *not* an implementation, but a textual description of a solution and its context.

Chapter 14 introduced already an important software pattern: the *mediator*. It takes care of the *decoupling* of the interaction between two or more components.

A *framework* ([Johnson97]) is *a set of computer code files that implement a reusable software solution for a complete but particular problem domain.* Important words in this definition are “implement” and “particular”: a framework is *code*, which only need some so-called “*hot spots*” to be filled in before it works. These hot spots are system-dependent parts of the software, such as: particular device drivers or user interface code.

A framework is much broader (“programming in the large”) than a software pattern (“programming in the small”). A framework typically contains several patterns, but a pattern doesn’t contain frameworks; a framework contains code, a pattern doesn’t. Frameworks are constructed in such a way that similar applications within the same domain can benefit from the same structures and abstractions, but may need re-implementation of the “hot spots” in the framework.

15.2. Monitor

The *monitor* is one of the older software patterns, developed for more complex mutual exclusion jobs than what the are RTOS primitives can deliver. Getting it working in a real-time application in a time-deterministic way, however, is not straigthforward, and its scalability is much worse than linear in the number of tasks, resources, and access conditions to be synchronized.

Semaphores (as well as all other locking mechanism discussed in the previous Chapters) are *primitive (“low-level”) tools*: programmers have to get the logic of using `sem_signal()` and `sem_wait()`

(Section 4.6.1) calls correct. One single mistake most often means an incorrectly functioning system. But, more importantly, using the locking primitives in this way also violates the *loose coupling* principle of good software design (Chapter 14): the synchronization is achieved by spreading lock function calls over the different tasks that take part in a mutual exclusion or any other synchronization. This distribution of the lock function calls makes later maintenance or updating more difficult and error-prone, because the maintainers should not forget to update *all* files in which the locks are used.

The solution that lies at the basis of the monitor pattern, is to avoid this spreading of locks by *keeping them all at one place*, protected in the internals of one single so-called "monitor" (or "*protected object*) that delivers the *serialization* service to client tasks. And, moreover, the monitor does this serialization in a quite specific fashion: it makes sure that, at any given time, *only one single client task* can execute any of the *set* of function calls that the monitor "protects." Or, in more modern object-oriented terminology: if one task calls a member function of the monitor, then it cannot be interrupted by another task that wants to call a member function of the same monitor. This aspect of mutual exclusion at the function calling level is the really new *synchronization primitive* that the monitor brings, in comparison with the classical IPC primitives found in traditional operating systems. (They provide mutual exclusion at the statement execution level.)

The monitor concept is another example of the *mediator* idea (Section 14.3):

- It mediates an intricate *synchronization* between several tasks.
- None of these tasks has to know the name or anything else about the other tasks involved in the synchronization.
- Part of the monitor could be the *protection of shared data* between the interacting tasks.

Obviously, a monitor needs locks internally to do the synchronization and the bookkeeping of the client tasks that request its services. But the advantage of the monitor is that it can keep all this bookkeeping inside its own code and that much of this bookkeeping must only be programmed once. So, the overall application code is easier to understand, predict, debug and maintain. The price paid for the convenience of a monitor is that: (i) it requires more complex (and hence slower) code; (ii) the exclusive execution within the monitor can lead to substantial delays for the client tasks that have to wait outside the monitor; (iii) using a monitor inside another monitor implies *nested* critical sections, and this leads to a deadlock sooner or later; and (iv) a monitor is difficult to distribute over a network, because it needs shared memory for the semaphores that it uses behind the screens.

The general monitor concept is not available as system call in any operating system, because, as will become clear later, the syntax of a OS function call is not sufficient to describe the full semantics of a general monitor. There are a number of runtime support libraries (such as the Mozilla NSPR library) and programming languages (such as Ada and Java) that offer monitors. But also in these cases, only a limited version of the monitor idea can be offered as a language or runtime primitive. Ideally, one would like to use a programming language syntax as shown in the pseudo-code skeleton of a monitor below:

```
monitor {  // begin of protected scope
mutex          monitor_mutex;// monitor-wide mutex to allow only one task
```

```
                             // to access the procedures below

data           shared_data;  // data structure protected by the monitor

mutex          cond1_mutex;  // first application-specific condition
pthread_cond_t cond1;        // variable and its mutex

mutex          cond2_mutex;  // second application-specific condition
pthread_cond_t cond2;        // variable and its mutex


// procedures that operate on the "shared_data":
procedure_1 {
  ...   // this procedure uses one or more of the application-specific
        // condition variables, to synchronize access to the
        // "shared_data"
}

procedure_2 {
  ...   // also this procedure uses one or more of the
        // application-specific condition variables
}

}  // end of protected scope
```

Of course, such an ideal syntax is not supported by operating systems. And the full implementation of a monitor in an operating system will be quite a bit different from the code above. But for the time being, the focus is on *meaning*, and not on *syntax*. So, a small example of an application that needs a monitor will hopefully help to make the discussion more concrete. The monitor object could be your bank account: the protected data structure is the unique resource (your money on the bank), and the method calls on that data structure are the classical things, such as redraw money, deposit money, check the available amount, etc. It is clear that clients should be allowed to access the bank account resource only one by one. For example, the husband checks the account, sees that it contains 1000 euros, and withdraws 800 of them. He should be guaranteed that, as soon as he was allowed to perform his first operation (check the account), he would be certain to proceed without another client coming in and changing the state of the account. For example, his wife also checking the account and withdrawing 800 euros. The monitor-wide mutex *`monitor_mutex`* is involved in allowing the husband to do his first operation, but disallowing his wife access to the monitor until he has finished is an *application-specific condition*. Of course, it is not difficult to replace the bank account scenario with any similar scenario in your real-time or embedded application where a shared resource must be accessed according to the monitor semantics. For example: an automatic clothing manufacturing cell has several machines that need material from the same textile sheet cutting machine; in order to guarantee color consistency of each single piece of clothing, each machine must be sure it can get all of the pieces it needs from the same textile batch.

Although far from complete, the ideal code fragment above does represent the essential aspects of applications that need the synchronization services of a monitor:

- The monitor has multiple “*access points*”: client tasks can call (i.e., ask to be allowed to execute) each of the procedures in the monitor, in any sequence, and at any time.

- Only *one client task* should, at any given time, effectively be allowed to execute some code of the procedures within the monitor, irrespective of how many clients have requested to do so. This mutual exclusion is the goal of the monitor-wide mutex *`monitor_mutex`*: it is used to allow only one task to enter what we will call the monitor's "activity zone."
- In addition to the monitor-wide mutex *`monitor_mutex`*, the different procedures in the monitor have *mutual critical sections*. Otherwise, putting them inside the same protected object would be an unnecessary complication. So, in general, a client task that is active in the monitor's "activity zone" (and, hence, runs in the critical section protected by *`monitor_mutex`*) *can block* on a task-specific condition. This facility to block when running inside of a critical section should sound familiar by now: it's the essence behind the introduction of *condition variables* on the operating system scene (Section 4.7). So, it comes as no surprise to see condition variables appear in the monitor procedures.
- Once a client task is in the monitor's "activity zone", it cannot be forced by other client tasks to leave that zone: it must leave it voluntarily. Otherwise, allowing the operating system or the runtime to force a task to stop anywhere in the code compromises the logical consistency of the monitor. Of course, the task can be pre-empted by the scheduler, and continue later on. However, this delays *all* client tasks of the monitor, because non of the other tasks is allowed to proceed in the activity zone. So, a monitor should be used with a lot of care in a real-time environment!
- So, the *`monitor_mutex`* protects the "activity zone" of the monitor. This zone is not explicitly visible in the code fragment above: it is a lock on a *task queue*, that would be behind the screens of a monitor implementation in a programming language: each client task that executes a method call on the object is blocked on this task queue, until it is allowed access to the "activity zone.". In a (real-time) OS context, this task queue cannot remain behind the screens, so the application programmer will have to make this queue visible (see below).
- *`monitor_mutex`* is a *generic* lock, that is part of every monitor structure, independently of the application that it protects. But the condition variables are *application-specific*. And that's the reason why a monitor can, in general, *not* be implemented as an operating system or runtime primitive: applications differ so much in their synchronization needs within the monitor procedures, that it is impossible to let users blindly fill in their procedures and condition variables as parameters in such a system call, with the guarantee that client tasks will only be active within the monitor one by one *and* according to the synchronization semantics of the application task. No, the monitor procedures must be carefully designed together, and this design cannot but make use of the error-prone and low-level operating system synchronization primitives discussed before. However, the advantage remains that all these primitives are used within one single programming scope.
- A monitor can be implemented as an abstract data type (i.e., using nothing but function calls on the monitor, as in the pseudo-code above), but also as an active component (i.e., in which each of the function calls above is replaced by a task that executes the function). The differences are that, in the active component version, the procedures run in the context of the tasks in the active monitor component, and the monitor clients could, in principle, continue doing something else after they have send their service request to the monitor. For example, in the bank account scenario above, the husband could use an internet banking application to prepare a set of bank account operations, send them to his bank in one batch operation, without waiting for the response from the bank server.

Because a task can block once it has been allowed into the monitor's activity zone, the monitor implementation becomes necessarily a bit more complex than the following simplistic solution that using nothing more but the monitor-wide *`monitor_mutex`*:

```
mutex_lock(&monitor_mutex);
```

```
switch (proc} {
  case (proc == procedure_1):
     ... // execute code of procedure_1
     break;
  case (proc == procedure_2):
     ... // execute code of procedure_2
     break;
}
mutex_unlock(&monitor_mutex);
```

This solution is simplistic, because, as said before, allowing a task to block in the *`monitor_mutex`* critical section can lead to deadlocks. So, the implementation of the monitor must make sure, “behind the screens” of what is visible in the code, that:

1. the task that is currently in the “activity zone”, leaves that zone to wait on a condition variable.
2. that task does *not* leave the monitor, because it is not yet finished with its monitor procedure and it holds a resource (i.e., lock) of the monitor which should not be exported to outside of the monitor.
3. it allows another task into its “activity zone.”

The section on condition variables (Section 4.7) showed that the condition variable primitive has exactly been introduced to allow a task to block within a critical section locked by a mutex. But in themselves, the normal condition variables are not sufficient: they work only within the code of one single procedure, and cannot span the scope of several procedures. Hence, the implementation of a monitor will be more complex than just using the *`monitor_mutex`* as the mutex of the required condition variables. The *`monitor_mutex`* and the condition variables must be *integrated*, in an application-specific way. And *that* is the reason why a general monitor does not exist in programming languages or runtimes.

Of course, not all applications need the full version of the monitor idea, so there exist various levels of functional complexity in the monitor concept. Not surprisingly, the more complex ones carry the highest cost in undeterministic timing behaviour. The following sections present monitors with increasing functional complexity.

15.2.1. Multi-procedure critical section

This is the simplest monitor, and it delivers only the service of *exclusive access* to its procedures. That means that only one client task can execute any of its procedures at the same time. So, the procedures in the monitor don’t have critical sections inside, but, on the contrary, they are within the critical section provided by the monitor wide *`monitor_mutex`*. So, the simplistic implementation above is sufficient, and this kind of monitor *can* be offered as a parameterized runtime primitive.

An example of such a monitor is the simplified version of the bank account: *each* operation on the bank account involves entering *and* leaving the monitor. So, one misses the “batch processing” functionality of the previously given example. Another example is a *command interpreter* of an embedded application that controls a machine: a client comes in with a request for a machine operation; such an operation request is typically translated in a sequence of multiple primitive actions on that machine, so all primitive actions in the client request should be executed before another client's request can be executed. However, this other client's request can already be *interpreted*, because that can happen outside of the monitor; the monitor is needed for the *execution* of the request, i.e., the unique and serialized access to the machine.

15.2.2. Semaphore-based monitor

The next level of monitor complexity comes when client tasks do have application-dependent synchronization needs, but these needs can be dealt with using *binary semaphores* only. This means that the synchronization condition on which task blocks in the monitor need not be checked explicitly: when the semaphore is signaled, the condition is *guaranteed* to be true. This kind of monitor is often called a *Hoare* monitor, after C.A.R. Hoare, who first described this semantics, [Hoare74]. Another name is *Mesa* monitor, after Xerox' graphical user interface language Mesa, in which it was first used, [LampsonRedell80]. The monitor has the so-called *Signal-and-sleep* semantics: the task that is in the monitor signals the condition semaphore, goes to sleep itself, while the signaled task runs immediately. The Hoare monitor is the kind of monitor that every object in Java offers to the programmer, via the `synchronized` access policy to its methods. While its *implementation* is a bit more complex (semaphores!) and time consuming (context switches!), its semantics are much simpler: by context-switching immediately to the signaled task, the monitor guarantees that this task knows that the condition is satisfied, because no other task in the monitor could have changed it. This semantics is only possible if the signaling task can indeed sleep immediately, i.e., when at the moment of signaling, it can leave the data structure in a consistent state. The waking and sleeping on the semaphore occurs *without* freeing the `monitor_mutex` mutex; so this synchronization is between two tasks *in* the monitor; a task outside of the monitor can only enter when all of the tasks that are already in, are waiting, or there are no tasks in the monitor.

An example is the classical producer-consumer buffer problem: the data structure in the monitor is a buffer, in which a producer task writes data, and from which a consumer client retrieves data. The semaphore is needed to signal and wait for the (binary!) condition that the buffer is empty or full:

```
monitor
{  // begin of monitor scope
const int BUFFER_CAPACITY = ...;
data buffer[BUFFER_CAPACITY];
data nextp, nextc;
int buffered_items = 0;
pthread_cond_t full = false;
pthread_cond_t empty = true;

produce_an_item()
{
  nextp = produce(...);
  if (buffered_items == BUFFER_CAPACITY) wait(full);
  // when going further here, there is certainly place in the buffer
  // and the consumer has set ‘buffered_items’ to its correct value
  buffer[buffered_items++] = nextp;
  signal(empty); // wake up some task waiting to consume an item
 }
consume_an_item()
{
  if (buffered_items == 0) wait(empty);
  // when going further here, something is certainly in the buffer
  // and the producer has set ‘buffered_items’ to its correct value
  nextc = buffer[--buffered_items];
  consume(nextc);
  signal(full); // wake up a producer
}

} // end of monitor scope
```

The checks for how many items are in the buffer take place in the critical section protected by the monitor-wide mutex. After the last signals in `produce_an_item()` and `consume_an_item()`, the producer or consumer task leaves the monitor, such that a new task can be allowed. This uses the monitor-wide mutes, and is not visible in the code; it is assumed to be done by the runtime.

This kind of monitor can also reasonably easy be offered as a parameterized primitive of a generally useful service, such as buffering.

15.2.3. Condition variable-based monitor

The most complex monitor allows its procedures to have synchronization needs that can only be dealt with using composite boolean expressions, such that condition variables are required. This kind of monitor is often called a *Hansen* monitor, after Per Brinch Hansen, who first described its semantics, [BrinchHansen73]. The semantics of the signaling is now *Signal-and-continue* : the task that is in the monitor and raises the signal continues, and the signaled task is put in a wait queue (within the monitor!). So, this task is not guaranteed of finding the condition fulfilled when it gets a chance to run again, and it should check that condition again. That’s the reason for the `while{}` loop in the code:

```
monitor
{  // begin of monitor scope
const int BUFFER_CAPACITY = ...;
data buffer[BUFFER_CAPACITY];
data nextp, nextc;
int buffered_items = 0;
pthread_cond_t full = false;
pthread_cond_t empty = true;

produce_an_item()
{
  nextp = produce();
  while (buffered_items == BUFFER_CAPACITY)
    { wait(full); }
    // the condition is _checked_, not just signaled
  buffer[buffered_items++] = nextp;
  signal(empty); // wake up someone waiting for an item
 }
consume_an_item()
{
  while (buffered_items == 0) wait(empty);
     // at this point, I m guaranteed to get an item
  nextc = buffer[--buffered_items];
  consume(nextc);
  signal(full); // wake up a producer
}
} //end of monitor scope
```

An example is a resource allocation system, such as the producer-consumer buffer above: the shared data structure is the resource, and the (de)allocation procedures check a lot of conditions before each client can get or release (part of) the resource.

This kind of monitor is very difficult to offer as a general parameterized primitive.

(TODO: give full code example. E.g. coordinating readers and writers example of [Nutt2000], p. 202, but with more complex conditions than the binary semaphores.)

15.3. Producer-Consumer

Section 14.3 introduced the general concept of a *mediator*; Section 15.2 explained how the *monitor mediator* works. And this Section applies the pattern to the very often used *Producer-Consumer* IPC between two tasks. The Producer-Consumer mediator is the object (data and methods) that helps task A to send data to task B, without having (i) to know anything about task B, and (ii) to worry about the implementation details of getting the data from A to B. Again, this *loose coupling* allows for easier maintenance and updates. For example, if task B is moved to another process or processor, the mediator can choose, internally, for a more appropriate type of communication and buffering, and neither A nor B have to be changed.

15.3.1. Terminology

This Section uses the following terminology:

- The *producer* is the task that wants to send the data.
- The *consumer* is the task that wants to receive the data.
- The *mediator* is the task (active) or object (passive) that producer and consumer use to perform their communication, without having to know each other.
- The data is also called the *message*.
- The asymmetry suggested by the terminology “producer” and “consumer” is not really relevant in the mediator pattern. So, both producer and consumer are called *clients* of the mediator.

- The mediator can be *persistent*, i.e., it is created once at start-up, and handles all requests during the lifetime of the interaction between both clients.
- The mediator can be *transient*, i.e., it is created each time a client issues a new request, and deleted as soon as the request has been handled.

15.3.2. Handling

Every line of code in a program is executed by one particular task (possibly the kernel). One says that the code "runs in the task's "*context*", using its stack, program counter, etc. In the method calls on the mediator object, it is not always clear or predictable which parts are executed in which context. For both unprotected and protected objects, everything that happens "in" the mediator is in fact executed using the stack and the context of one of the clients. One discriminates between the *synchronous* and *asynchronous* parts of every call of a mediator method:

- *Synchronous.* This is the part that executes *most definitely* in the context of the calling client. "Synchronous" here means: "activated by code in the method call" that the client performs. *Asynchronous.* This part does *not necessarily* run in the context of the calling client (but that remains possible), because it is executed "asynchronously". That means, it is not directly activated by code in the client call, but by other methods of the mediator. These other method calls can, for example, be activated by the other client of the mediator. A typical example: the synchronous part gets the data from the producer to the mediator protected object, where it stays until the consumer asks for it later on ("asynchronously").

Every client call involves, in general, three distinct handlers (or "services") by the mediator: synchronous, asynchronous, and completion handling:

- *Synchronous handling* is that part of the interaction that is done in the client's method call: the client changes some mediator data structure that remembers that this call has taken place and that it needs further handling (in other words, it makes the producer-consumer data exchange "pending"), and possibly also copies the data needed for this further handling. In general, this synchronous part involves some locks on protected data structures, and hence possibly blocks the calling thread.
- *Asynchronous handling.* The mediator usually has to do more work than the message copying and bookkeeping in the synchronous part: the message must effectively be delivered to a consumer; the buffers must be updated according to incoming priority and cancellation request; an event that has fired has to be serviced; etc. How exactly the further handling is done depends on the type of the mediator:
  - *The mediator is a passive object (unprotected or protected object).* In this case, one of the interaction initiating client tasks executes *all* the asynchronous handling that is pending in the mediator. Not only its own handling, but that of all pending requests. So, for this client, there is no real distinction between the synchronous and asynchronous handling parts.
  - *The mediator is a task (active object, or component).* The IPC initiating client continues after the synchronous handle finishes, i.e., it has put all data for further handling in an appropriate buffer, and the mediator further processes this data later on, in its own context.

- *Completion handling* is an optional third handling part that clients can ask for. For example: a producer sends new data to a consumer only as soon as it knows that the consumer is ready to accept the new data. Or, the producer blocks until the mediator has completed the interaction, after which the producer is woken up by the mediator.

  Completion is performed *after* the asynchronous handling, and in the context of the task that just finished executing the last asynchronous handling. Completion could involve the mediator accessing data structures of the client that has registered the completion. Hence, completion requires more careful coding from the application programmer, because it runs asynchronously and hence the context is probably not the one of the registering client. An additional concern for the application programmer is that it is not always straightforward to guarantee that, at the time of completion, the called object still exists. In summary, treat a completion handler as if it were an interrupt handler: it should never do something that can block.

The client (or every task authorized to do it for the client) has to register *explicitly* a completion function (or "call-back function") with the mediator. Synchronous and asynchronous handlings must not be registered in the case of a Producer-Consumer mediator, because they are the default send and receive methods of the Producer-Consumer mediator. (However, other types of mediators could require explicit registering of client-specific functions for *all* handlers.) The mediator calls the registered completion function at the appropriate moment, i.e., after all asynchronous handling has been done.

Note that (i) asynchronous and completion handling need not be present in every mediator; (ii) asynchronous handling must always come after the synchronous handling; and (iii) completion must always come after asynchronous handling.

15.3.3. Data buffering

The mediator can buffer the protected message data in various ways, as explained in Chapter 5.

15.3.4. Data access

Accessing the data of the message object in the mediator can happen in various ways, each with a different trade-off between blocking and data protection:

- *Unprotected object.* The data of the message is directly accessed in the IPC function calls, i.e., it is *shared memory* in the form of a global variable. This allows for the most efficient data access, with the shortest amount of blocking, but it is only a viable IPC mechanism if producer and consumer are *guaranteed* not to access the data simultaneously. Indeed, in general, these IPC calls on an unprotected mediator object are *not thread safe* because the object contains data that is shared between consumer and producer. Hence, this approach is only viable in fully deterministic cases, such as an ISR-DSR combination.
- *Protected object* (Also called "monitor" in some literature.) The data of the message is not directly accessible to producer and consumer. They must use access method calls ("read" and "write"), which

are *serialized* within the mediator by some kind of mutual exclusion lock (mutex, semaphore, \dots) around the data. This allows (but does not automatically guarantee!) safe access to the message data, but producer and consumer can block on the lock.

- *Active object.* This is conceptually the same as a protected object, but with one important difference: the mediator has its own thread.
- *Component.* The protected or active objects are a good solution in a system in which producers and consumers know which objects to use. Modern software systems become more dynamic and more distributed, and having to know the identity of all services in the system becomes a scaling bottle-neck. Therefore, the concept of components has been introduced: they have a protected object inside, but have extra functionality to work together with a *name server* that offers run-time and network-transparant bindings between components.

15.4. Events

The event pattern describes how to synchronize activities running in different task, with very loose coupling between the tasks. The event pattern is applicable to the RTOS primitives *signal* (Section 4.3) and *interrupt* (Section 3.3), with only minor adaptations:

- Tasks must *register* to get notified about events and interrupts, while (in the POSIX semantics, at least) they have to explicitly *de-register* from every signal they don't want to receive.
- The synchronous handling of interrupts is initiated by the *hardware*, and not by another software task.
- Events can carry any form of data, while signals, and interrupts are data-less triggers. Or, almost data-less: they can carry information about, for example, the time or the cause of the triggering.

15.4.1. Semantics

The event pattern has a lot in common with the Producer-Consumer pattern, but its emphasis is on *synchronization of the tasks' activities* and not on *data exchange*. In any case, this discussion re-uses as much material of the Producer-Consumer pattern as possible. The semantics of a general event are as follows:

- *(De-)Registration* of a *listener* function. Registration is a *configuration time* activity, and is not part of the event's interaction itself. At registration, a task gives (a reference to) a function call (the "listener") which must be called as *synchronous or asynchronous handler* whent the event *fires*. Whether the listener is synchronous or asynchronous is again a configuration option. When called, the listener gets information about which event has caused it to run.

  Multiple processes can register their listeners with the same event. And the same listener can be registered with several events.

- *(De-)Registration* of a *completion* function. This is technically similar to listener registration, but functionally different: the completor is called by the event mediator when all synchronous and asynchronous activities for this event have finished. The task that registers a completion function need not be the one that registers the listener. And a task doesn't have to register in order to be allowed to fire an event.
- *Firing.*
- *Firing.* A task fires the event, i.e., it executes a method call of the mediator, that performs the *synchronous handling* of the event, and puts the asynchronous and completion handling in the pending queue for the fired event. (All this requires synchronized access to the mediator's bookkeeping data structures.)
- *Guard.* Whether a firing event really sets the handling in acion or not, can be made dependent on a *guard*. This can be any Boolean expression, which prevents the firing if its evaluation returns “false”. The evaluation happens instantaneously, at the fire time of the event, i.e., in the synchronous handling by the mediator.

  Be careful with guards: they are a too powerful mechanism for scalable and deterministic software systems. Having a Boolean decide about whether or not to do something is often a sign of a bad design: it's much cleaner to have this “state dependency” inside the mediator, by providing it with the *State Machine* mechanism (Section 15.5).

- *Handling.* This covers the asynchronous and completion parts of the event. So, handling calls the registered listeners and/or completion functions.

15.4.2. Policies

Events can have multiple *policies* on top of the above-mentioned mechanism. All of the following ones can be combined:

- *Queue pending events.* The event mediator can have a queue for each listener, in which it drops every fired event, at synchronous handling. So, no events are lost when a new one arrives, while the synchronous processing is still busy with a previous event.
- *Prioritize listeners and completors.* This allows to influence the order in which they are executed.
- *One-off execution of listeners and completors.* This means that on each fired event, only one of the registered listeners and completors is executed.

15.4.3. Composite events

Often, a task wants to be notified not just when one particular event has been fired, but whenever a logical “AND” or “OR” combination of several events has occurred. Such a composite event `C` could be implemented by a mediator event between the process `P` and the two events `A` and `B`. For example, for the AND composite event:

- The task `P` registers its listener and completion function with `C`.
- `C` registers listeners for both `A` and `B` but registers no completion handler.
- The listeners for `A` and `B` look like this:

  ```
  if (A_has_fired and B_has_fired)
     clear A_has_fired
     clear B_has_fired
     fire C
  ```

  This code runs internally in the mediator of `C` hence it can access the flags *`A_has_fired`* and *`B_has_fired`* atomically, without needing the overhead of a critical section.

Of course, any Boolean expression of events can be implemented as a composite event. Whether or not to provide a separate composite event for a specific Boolean expression is an efficiency trade-off: writing a new object, versus introducing multiple levels of the simple AND and OR composite events.

15.4.4. Loop with asynchronous exit event

A task `P` cyclically runs a function `F` which it must execute whenever event `A` occurs. But whenever event `B` occurs, the task should exit the loop around `F`. `B` can be asynchronous to the execution of the loop, so it's best to let `P` finish the loop, and only then exit. This can be done by having the process wait for the "OR" of both events, and then take appropriate action (loop or exit). Here follows a possible implementation, where the composite event signals a condition variable:

```
Task P:                        Composite event listener:

while (1) {                    if (A_has_fired or B_has_fired)
  wait_on_condition(A_OR_B);      broadcast(A_OR_B);
  if (A) F;
  if (B) exit;
  }
```

The exit is done "synchronously", i.e., it never interrupts the loop function `F`. As a result, the process comes out of the loop in a predictable state.

15.4.5. Some caution

Let's conclude this Section with a critical note by Per Brinch Hansen on event variables [BrinchHansen73]: "Event operations force the programmer to be aware of the relative speeds of the sending and receiving processes." And: "We must therefore conclude that event variables of the previous type are impractical for system design. The effect of an interaction between two processes must be independent of the speed at which it is carried out." He was talking about using events as a *general* multi-tasking synchronization primitive, replacing the synchronizations of Chapter 4. And in that context, his remarks are very valid (and he suggested *condition variables*, Section 4.7). But there are situations where the synchronization is *not* time-dependent; for example, the feedback control example in Chapter 16.

15.5. State Machines

A state machine is a common way to give structure to the execution of computer tasks: a task can be in a number of possible *states*, performing a *particular function* in each of these states, and making a transition to another state caused by either an external event or internal state logic. So, a state machine is the appropriate pattern for an application in which different modes of control are to be available, and the transitions between these mode is triggered by events.

This Section discusses *object* state machines: the state machine doesn't describe the classical *process* of actions triggered by events, but it allows an *object to change its behaviour* through events. The software engineering advantage of the object-based approach to state machines is that the internals of the states need not be exported outside of the object. This Section describes the mechanism of one particular type of state machine, where the design goal is to maximize determinism and semantic unambiguity, at the cost of ultimate generality. The execution of the above-mentioned state functionality requires, in general, *finite amounts of time*, while the mathematical state machine reacts in zero time. No software approach can guarantee such zero-time execution, but the presented object state machine does guarantee that all state functions are executed atomically within the context of the state machine, i.e., state functions are properly serialized with state transitions.

(TODO: code examples; Hierarchical State Machines eventhelix example (http://www.eventhelix.com/RealtimeMantra/HierarchicalStateMachine.htm)?;)

15.5.1. Semantics

An object state machine is a composite class that manages the following data:

- One *class* for each state in the state machine. It contains the functions to be called in the state, as discussed below.
- *Events* to make the object *transition* to other states.
- A *graph*, to represent the structure of the state machine: a node is a state class, and an edge is a transition between states.

The choice for a graph object corresponds to the choice of a *persistent* state machine mediator: the graph persistently stores the information of transitions and related events, such that this information is directly available and no time is lost creating or deleting state objects. This is an example of the classical trade-off between computation cost and storage cost of performing the same functionality.

Figure 15-1. General structure of a state.

Figure Figure 15-1 shows the general structure of a state:

- *Entry.* This function runs when the object first enters a state. If a state is implemented as a transient object, this would be the state object's constructor. The last thing the entry function does is to call the state function.
- *State function.* The state object runs this function after the entry. The state function is guaranteed to be executed *atomically* (i.e., without interruption) *within the context of the object's state machine*. That is, no state transitions can happen when the function is running.

  The state function can be an *action* or an *activity*:

  - *Action.* The state function runs once, performs a certain "action" (such as setting an output), and then runs the exit function (see below).
  - *Activity.* The state function runs in a loop, from which it exits when it receives the "abort" event, or when it decides itself to exit from its loop.

- *Exit.* This function runs when the object is about to transition to another state. For a transient object, it would be the object's destructor.

  The exit function calls the state machine object (with as parameters the current state and the event that has caused the transition) to load the next state information in the state object, and (optionally) fires an event that signals the state exit. Loading the next state means that new entry, state and exit functions are filled in in the corresponding data structure of the state machine object.

  When the next state is equal to the current state, the object goes directly to the state function, without executing the entry function again.

This mechanism does not represent the most general form of state machine: the atomicity of the state function (action as well as activity) is a restriction on the generality, but this serialization of state function execution and state transitioning adds a lot to the *determinism* of the state machine. If the state function is not guaranteed to run until completion, the object could end up with unpredictable and inconsistent values of some variables. The atomicity is only guaranteed within the context of the running task, and not within the whole software system.

The presented mechanism can represent both *Moore* and *Mealy* state machines. If the state function is an activity, the state machine is a Moore machine. Action (or "(discrete) change") is associated to a transition, and in that case, the state machine is a Mealy state machine, [Mealy55]. So, Moore machines are appropriate for continuous, non zero-time activity (such as software objects), and Mealy machines for discrete changes (such as electronic circuits).

15.5.2. Implementation with events

The execution of a state machine can be implemented on top of the event mechanism of Section 15.4. The state machine object has an event object for each of its transition events, and it has a data structure that stores the entry, state and exit functions of the currently active state. The event that causes a state transition has been initialized as follows:

- Its listener executes the current state's exit function.
- Its completer executes the new state's entry function (unless the new state is the same as the old state), as well as its state function.

The listener and completer select the right functions from the information in the state machine graph, and from the identity of the current state.

In principle, both actions (i.e., exit and entry functions) could be done by the listener. But if the event causes more things than just a state transition in a state object, it could be interesting to have all this event's listeners executed before the completer executes any of the state entry functions.

The above-mentioned run-time registration of listeners and completer is not always a good idea, because registration involves a lot of linked list operations. An alternative is to have the state machine listen to all events, and let its listener call the corresponding state listeners and completers.

The advantage of using events to trigger transitions is, that the knowledge of to which next state to transition at exit, is not stored in the current state, but in the state machine object's graph structure. In this sense, that state machine object is a mediator between the different states.

15.6. Execution Engine

(TODO: *sequencing* of non-distributed but non-linear activities;)

The *Execution Engine* is a pattern that takes care of *activation* and *configuration* of software components:

- Activating components respecting their individual timing specifications.
- Run-time configuration of components.

The Execution Engine is a mediator object (Section 14.3) in the sense that it decouples the activation and configuration synchronization of several components. Unlike previously discussed mediators, it doesn't take care about any *data exchange* or *mutual exclusion synchronization* between the components.

By localizing the *activation logic* of a complete application in one single mediator, the system is much easier to understand, program, adapt, and make deterministic. The core of the Execution Engine can be a *finite state machine*, whose outputs are triggers for the other components; for the pacing of its state machine, the Execution Engine relies on basic timer functionality of the RTOS on which it is implemented.

15.7. Distributed IPC

The important domain of *component distribution and communication* has already been developed quite extensively. Douglas C. Schmidt's free software projects ACE (http://www.cs.wustl.edu/~schmidt/ACE.html) (*Adaptive Communication Environment*) and TAO (http://www.cs.wustl.edu/~schmidt/TAO.html) (*The ACE Orb*) are primary references. This work has been an important basis for the specification of *Real-Time CORBA*, Section 1.5.8.

There is sufficient documentation and code available on-line, so this text will not go into more detail. Especially because distributed IPC is inherently not a hard real-time system.

(TODO: more details; example with real-time or embedded relevance: *DAIS* (Data Acquisition from Industrial Systems Specification, OMG group effort for large-scale data acquisition);)

15.8. Transactions

Important concept in the context of databases: *ACID* (http://www.cis.temple.edu/~ingargio/old/cis307s01/readings/transaction.html) (Atomicity, Consistency, Isolation/Serializability, Durability). In that form, it is too heavy for real-time systems, which often interact with a real world, in which it can impossibly undo actions. But an more realistic sub-primitive is the *Two-Phase Commit* (TPC) pattern for atomic transactions in a Distributed System, [Gray78], [Galli2000], [BurnsWellings2001], p.390: two tasks want to be sure that both of them agree on a particular action being done, and the TPC guarantees that the action is completely done or completely undone. The first of the two phases is the *negotation and set-up phase*, and the second phase is the *execution phase ("commit")*.

(TODO: more details; example with real-time relevance;)

Chapter 16. Design example: “control”

“Control” is a very mature and broad domain, with thousands of research publications every year. Most of these publications deal with new *applications* of existing concepts and technology, or with improved *functionality* of existing aproaches. There is almost no evolution anymore in the *fundamentals* of the technology. But this lack of evolution is not perceived as a problem, because the fundamentals are mature and have proven to work. This means control is an exquisite subject to define Software Patterns for, Chapter 15.

This Chapter describes these Patterns, as far as they are relevant for the *real-time* software engineering aspects of the problem. It presents (a design for) a generic, hard real-time control framework, making use of the decoupling ideas and other Software Patterns introduced in the previous Chapters. A similar discussion could be held for other mature application areas, such as telecommunication.

(TODO: are there other hard real-time areas besides control and telecom? Is telecom really hard real-time? Or is its hard real-time functionality only the signal processing, which we take as part of the generic control pattern?)

The first message of this Chapter is that many complex hard real-time systems can be built using only an amazingly small set of the primitives offered by a typical RTOS. (The design presented in this Chapter can even run *without* operating system.) This fact often comes as a surprise to students or newcomers in the field, because they tend to come up with systems that have separate tasks for every piece of functionality in the system, and that need complex IPC, driven as they are by their eagerness to use the largest possible set of the RTOS primitives they’ve learned in the classroom. So, also in real-time and embedded application programming, simplicity of design is the signature of the real craftsman.

The second message is inspired by the observation that experienced designers in *every* particular application domain introduce a lot of *structure* in the way they solve the application problems. They do this most often *implicitly*. So, the message is to first make explicit the largest form of structure that is generic for the application domain, and then use it to build the *infrastructure* parts in your design. The rule of thumb is that structure *always* leads to efficiency gains, in design, in implementation, and in documentation.

The third message is to document and localize the *“hot spots”* in your design. That is, those parts that will have to be changed whenever the application is ported to new hardware and a new operating system. This Chapter calls them the *device interface* and the *operating system interface*.

16.1. What is control?

This Chapter uses “control” to illustrate the above-mentioned messages. It interprets the concept of control quite broadly: the presented framework covers various domains, known under names such as:

- *Pure data acquisition*, as implemented by Comedi (Section 7.4).
- *Extended data acquisition and generation*, with pre- and post-processing of the signals. For example, applications which must calibrate equipment against standards, send specific pulse trains, detect peaks and abrupt changes, etc. The presented design can be seen as an extension to Comedi, adding signal processing and feedback control funtionality.
- *Waveform generation*, which is the same as the generation application above.
- *Programmable Logic Control (PLC)*, i.e., the “primitive” form of control in which all inputs are read first and stored in a buffer, then a set of (possibly unrelated) functions are run in sequence, each producing one or more values in the output buffer, which is finally written to the peripherals “in block.” PLC functionality is present in most machine tools, to manage discrete actions, such as closing valves, setting LEDs, and even simple control loops such as *PID* (Proportional-Integrative-Derivative).
- *Feedforward/feedback control*, such as in robotics or other mechatronic systems.
- *Observation*, i.e., measured signals are monitored, and specific patterns in the signals are detected and reacted to.
- *Estimation*, i.e., the measured raw signals are processed, and estimates of non directly measurable quantities are derived from them. Observation and estimation are often used as synonyms; this text will do that too.
- *Signal processing*: still another name for all the applications mentioned above (i.e., those that don’t drive outputs based on measured inputs).

16.2. Functional components

The presented design is limited to the *common real-time (infra)structure* needed by all these applications. Application-specific *functionality* must be implemented on top of it, via *“plug-ins.”* This Section presents the (application-independent) *functional* parts of the generic control system. (Some of the above-mentioned application areas don’t need all of these components.) Each functional component has a specific goal in the overall control (i.e., it runs an application-specific *algorithm*), and the interfaces between the parts are small and well defined.

An interface consists of: (i) data structures; (ii) function calls; and (iii) events (Section 15.4). Data structures and function calls can be considered as one single part of the interface, by assuming that each access to the data takes place through a function call. The event information in the interface specifies for which events the component has a “listener” (without saying explicitly what the listener does), and which other events it can “fire”.

The following Section discusses the *infrastructural* parts of the design, i.e., those that support the functional components in their actions, but contain no application-specific functionality themselves. The functional components are:

- *Scanner*: measures signals on interface cards.
- *Actuator*: writes setpoints to interface cards.
- *Generator*: generates signal setpoints. It supports *hybrid signals*, i.e., discrete signals (“pulses”), analog signals (in sampled form, of course), as well as discrete switches between analog signal forms. In its signal generation, it can make use of the data that other components have available. In control theory, one calls the functionality offered by the Generator “*feedforward*” and/or “*setpoint generation*.”
- *Observer*: reads Generator and Scanner results, and calculates estimates on these data. Lots of application-dependent forms of data observation exist, known under names such as “filtering,” “transformations,” “data reduction,” “classification,” etc.
- *Controller*: reads Generator, Scanner and Observer, and calculates setpoints for the Actuator, in its “*control algorithm*.”

These are the *functional components*, i.e., the components of which application programmers see the plug-in interface, and for which they must provide functional contents, in the form of the signal generation or processing algorithms of their application.

When they are present, these functional components are *always* connected according to the same structure, depicted in Figure 16-1. This figure shows the functional components (and the infrastructural components discussed in the following Section) as rectangular boxes. They interact through Producer-Consumer *mediators* (Section 15.3), depicted by ovals.

Figure 16-1. Structure of generic control application.

16.3. Infrastructural components

The design also needs some *infrastructural components*, that run “behind the screens” in order to support the functional components, but that don’t execute any application-specific algorithms. These components are:

- *Execution Engine*: is responsible for *activation* and *configuration*:
  - activating the functional components, respecting their individual timing specifications.
  - run-time configuration of the functional components.

  This is the only component that knows how the other components should interact, and it triggers other components to execute their functionality. By localizing the *application logic* in one single component, the system is much easier to understand, program, adapt, and make deterministic. The core of the Execution Engine is a finite state machine, whose outputs are triggers for the other components; for the pacing of its state machine, the Execution Engine relies on basic timer functionality of the RTOS.
- *Command Interpreter*: this is *not* a hard real-time component, because it receives commands (configuration, action specification, etc.) from *user space* (in whatever *protocol* the application uses), parses them, checks their consistency, fills in the configuration data structures for the other

components, and signals the Execution Engine when a complete and consistent new specification for the real-time system is available. It has to make sure that its communication with the real-time Execution Engine is *atomic*: either the whole new specification is transferred, or nothing. “*Swinging buffers*” (Section 5.5) are a possible RTOS IPC primitive to implement this atomicity.

- *Reporter*: collects the data that the other components want to send to the user, and takes care of the transmission.
- *HeartBeat*: this component handles the timer ticks from the operating system, and derives a “virtual system time” from it. The Execution Engine asks the HeartBeat to activate some components (i.e., to fire appropriate events) at particular virtual time instants.

16.4. Design

This Section explains the design decisions behind the structure of Figure 16-1.

One major design choice is to introduce *maximum decoupling* between components. This is achieved in various ways:

- Dividing the whole application into components with a *minimal amount of interactions.* The whole system has simple “Producer-Consumer” interactions, and the interaction graph has *no loops*. Execution Engine, Generator, Observer and Controller can be designed fully independently of RTOS and user, because they interact only with Scanner, Actuator, Reporter and HeartBeat.
- Minimizing the *RTOS primitives* that each component needs. The HeartBeat needs input from the timer of the RTOS, and the mediators need locks to sequence the access to the interaction data they encapsulate. They do need more, only in case the system is distributed over a network, by cutting a mediator in two.
- *Localizing* each of the component-component interactions into a mediator object.
- *Localizing* the RTOS interaction in the so-called *operating system interface*.
- *Localizing* the hardware interaction in the so-called *device interface*, so that it can be ported to other platforms, or to user space, running on virtual hardware, used for example for simulation or non-real-time signal processing.
- Using events allows the system to not rely at all on the *scheduler* of the RTOS (see next Section).

Another design choice is to provide a design that can be flexibly configured, going from everything running as one single task, even without an operating system, to a system where each component runs an a separate processor. This design goal has been reached as follows:

- *Events.* This is one of the best decoupling mechanism to use at all possible levels of distribution (*if* the application allows it, Section 15.4.5). Events encompass hardware and software interrupts, exceptions, state machine actions, CORBA events, etc.

- *Mediators.* Since all information about an interaction is localized in these mediators, distributing the mediators is all it takes to distribute the application. Everyhting outside of the mediators remains unchanged.

16.5. Implementation

A full control application may seem quite complex at first. But the structure of the application, and the design decisions explained in the previous Section, make a very simple and efficient implementation possible on a single-processor system.

The key behind the implementation is that the *structure* of the application is a *loopless graph*. This means that there is a deterministic way to *serialize* the whole execution of the control system. There are two natural orders, *push*, and *pull*,. Push means that the execution starts at the Scanner component, that reads the hardware, and produces data for its mediator, that mediator then uses these “inputs” to trigger the Observer, and the Generator. Then, the Controller works and finally the Actuator. After they have done their job, the Command Interpreter and Execution Engine are executed. Pull is the same thing, in reverse order, starting from the Actuator.

All this “execution” is nothing else but running the event handlers of the HeartBeat virtual tick event: all functionality of all components is registered as listeners to that event. The order of the execution of the listeners corresponds to the natural order in the presented control design.

The serial execution above could also be executed in one single combination of ISR and DSR, (Section 3.4), where the ISR is triggered by the hardware timer of the system. So, in principle, this implementation doesn’t even need an operating system, and is appropriate for embedded implementations that require little flexibility.

The fact that all execution can nicely be serialized deterministically allows to use *unprotected objects* in the mediators (Section 15.3.4), again improving efficiency because no locking is needed.

IV. Tips and tricks

This last Part is a collection of more or less unconnected tips and tricks, that can help application programmers to solve many of those little annoying problems that show up in a software project. Of course, the emphasis is again on real-time and embedded applications.

Chapter 17. Tips and tricks

TODO: memory barriers; exception handling: recover, hang up or crash, error recovery vs die on the spot; time stamps; garbage collection vs fixed size chunks vs static allocation;

17.1. Tasks

The term “*thread pool*” is often used in the context of servers that have to process lots of service requests coming in asynchronously from client tasks. The term makes one think about a company that hires workers when it needs them and gives them a job to do. This “active way of distributing jobs” is a rather unfortunate analogy to think of programs: you shouldn’t be thinking about “giving the threads work to do”, but about “announcing that there is work to do”. The threads will then pick up that work when they are ready. The Producer-Consumer mediator model is the way to go here. A request comes in, the producer puts it on a queue, and a consumer takes it off that queue and processes it. Consumer threads block when there is nothing to do, and they wake up and work when jobs become available.

The thread pool example above is one of those many occasions where programmers create a “manager” task: that manager takes all the decisions, such as *actively* deciding when a certain task has to start and stop. But trying to start and stop threads from an external task is error prone Section 2.2. Trying to *delete* another task is even more dangerous: there is no way you can determine when another task is not involved anymore in IPC with other tasks, or when it has released all of the locks it holds on shared resources.

Determining a correct stack size for your tasks is often a difficult job. If you have the possibility to experiment with your application in realistic and worst-case environments, the following trick can help you out:

- Allocate a quite large stack size for a task.
- At creation of the task, fill the stack with a regular pattern, such as “123412341234…”.
- At the end of your test run, check how much of the regular pattern has been overwritten. This is a *lower bound* in the stack size this particular task should get.

17.2. Signals

Signals and threads do not mix well. A lot of programmers start out by writing their code under the mistaken assumption that they can set a signal handler for each thread; but signals operate on the *process*, i.e., all threads receive all signals. One can block or unblock signals on a thread-by-thread basis, but this is not the same thing.

However, in Linux each thread is a process, and has its own signal handling. Linux executes signal handlers in the `ret_from_inter` action (see `arch/xyz/kernel/entry.S`, with `xyz` the name of a particular CPU architecture).

If you have to deal with signals, the best you can do is to create a special signal handling thread: its sole purpose is to handle signals for the entire process. This thread should loop calling `sigwait()`, and all threads (including the one that calls sigwait) block the signals you are interested in. This allows your system to deal with signals synchronously.

Sending signals to other threads within your own process is not a friendly thing to do, unless you are careful with signal masks.

Using `sigwait()` and installing signals handlers for the signals you are sigwaiting for is a bad idea: one signal will generate two reactions in your application, and these reactions are hard to synchronize.

Let threads sleep on time or condition variables only: this makes their actions on wake-up deterministic. So avoid `pthread_suspend_np()` and `pthread_wakeup_np()`. POSIX didn't include these calls because they are too easy to lead to inconsistent system, but UNIX98 has them.

17.3. Condition variables

Don't mistake the (POSIX, Section 1.5.1) *condition variable* for a *logical condition*: the condition variable act like a signal, in that it is only the *notification* that some logical condition *might* be changed. When coming out of the blocking, the task should check the logical condition again, because the signaling through the condition variable doesn't guarantee anything about the value of the logical condition. Have a look at the example in Section 4.7.

17.4. Locks

Application programmers are responsible for acquiring and releasing locks; they cannot expect much help from programming tools or from the operating system to use locks efficiently and effectively. It is indeed *very* difficult to interpret automatically the *purpose* of a lock, i.e., locks are really part of the *semantics* of a program, and much less of its *syntax*. Moreover, locks work only when *all* tasks that access the resource obey the (non-enforceable) lock: any task can just decide not to check the lock and access the resource, without the operating system or other tasks being able to prevent it.

The programmer should think about the following when using locks:

- Make sure the sections protected by locks are as short as possible, *and* remain buried in the operating system code, or in *objects* (encapsulated data types) in the application's support libraries or components.

- Make sure interrupt routines do not share locks with non-interrupt code. If this condition is not satisfied, the interrupt routine can block on the lock, or the non-interrupt task that sets a lock can never be sure that an interrupt routine will not enter its critical section. Here is an example that leads to a deadlock:

```
lock lock_A;
        ...
        // in task A:
        get_lock(lock_A);
        ...
                // Here, an interrupt routine comes in
                // and tries to get the same lock:
                get_lock(lock_A);
                ...
```

- Use locks only *locally* (i.e., in at most two tasks, and without nesting) and *focused* (i.e., use one lock for one purpose only, and give it a relevant name). Although this is not a strict requirement, violating it leads to complex code, which is error-prone and difficult to maintain and extend.
- Place lock and protected data in the same data structure. They *really* belong together, to form a “protected object”.
- If interrupt routines and kernel or user tasks share critical sections (which they shouldn’t!), the latter ones should *disable interrupts* when entering the critical section. Again, many processors make this kind of combined operation available (test and set lock, disable interrupts) in an atomic version. But be aware of its cost!
- Never use a recursive mutex with condition variables because the implicit unlock performed for a `pthread_cond_wait()` or `pthread_cond_timedwait()` might not actually release the mutex. In that case, no other thread can satisfy the condition of the predicate.

17.5. Interrupts

The correct place to call `request_irq()` is when the device is first opened, before the hardware is instructed to generate interrupts. The place to call `free_irq()` is the last time the device is closed, after the hardware is told not to interrupt the processor any more. The disadvantage of this technique is that you need to keep a per-device open count. Using the module count isn’t enough if you control two or more devices from the same module. . . .

In some operating systems, interrupt code runs on the stack of whatever task was running when the interrupt happened. This complicates the programmer’s job of choosing an appropriate stack size for tasks.

17.6. Memory

Some peripheral devices use *Direct Memory Access* (DMA) (Section 5.1) Often, it's a practical problem to get enough *contiguous* memory, e.g., the device expects to be able to dump its data to 2 megabytes of RAM without "holes" in the address range, while the operating system doesn't have such a big chunk of physical RAM.

One way to solve this problem is to set aside a part of the available RAM at boot time, Section 6.2.1. This means that the operating system will not use that RAM for anything, such that an application can use it. Of course, if your application has several tasks that want to use this RAM, you have to do the memory management yourself. As an example, the Linux operating system allows a boot parameter option as follows:

```
linux mem=128M
```

indicating that only 128 of the available megabytes will be used by the operating system. Boot loaders, such as lilo or grub, have similar configuration options.

Another approach is *scatter/gather DMA*: the operating system divides the physically non-contiguous DMA buffer into a *list* with entries that contain (i) a pointer to a physical page, and (ii) the amount of contiguous RAM available at that place. Typically, all these physical pages have the default size of your operating system, except probably the first and the last. To initiate the DMA, you load the first pointer/size pair from the list into the DMA controller, and program it to issue an *interrupt* (Section 3.3) when the DMA is done. Then, in the interrupt handler, you re-initiate the DMA with the next pair of values from the list. This is repeated until the list is exhausted.

17.7. Design

Don't make use of platform-specific function calls or data structures: use standards (e.g.; POSIX), or encapsulate platform-specific code in libraries with a neutral API. Difficult!

17.8. Programming

The *`volatile`* keyword is an important feature of the C compiler for real-time and embedded systems. These systems most often interact with peripheral hardware, via *memory-mapped I/O*. That means that the harware's registers are read from, or written to, as if they were a couple of bytes in the normal RAM of the system. Typically, some registers of the hardware are always read, some others always written. And many peripheral devices use the same register for subsequent reads or writes. Hence, the following code fragment is typical for such an operation:

```
char *writereg = 0xFF20000;
char byte1, byte2;

...
```

```
*writereg = byte1;
*writereg = byte2;
...
```

Most compilers are doing lots of optimizations behind the screens. And they will “optimize away” the code above to one single write, because their reasoning is that writing to the same variable twice in a row amounts to the same thing as only writing the last value; indeed, the first write is overwritten immediately. But this is not what one wants to access peripheral hardware registers. To prevent the compiler from optmizing away these multiple write, one should use the *`volatile`* qualifier in front of the *`writereg`*.

Default initialization: the C standard says that static integers are automatically initialized to zeros. This is often used by programmers as an excuse not the initialize their variables explicitly. The arguments being that (i) in standard C, and (ii) explicit initialization requires a couple more bytes in the binary. Nevertheless, explicit initialization does help other coders to better understand your code. And remember: this implicit initialization is nothing but a *syntactic* support from the compiler, which may well lead to *semantic* errors! For example, your code compiles because a condition integers has gotten a value (zero), but the logic of your application requires that it would have been initialized to one.

For embedded applications, making the binaries of the loaded code as small as possible is important. Normal compilation results in quite some “overhead” in this respect, such as symbolic data, etc. There exist various ways to make binary code smaller: **strip**; using C or GUI libraries that are especially designed for embedding, such as ; etc. For example, BusyBox (http://www.busybox.net) is a replacement for most of the utilities one usually finds in the GNU `fileutils`, `shellutils`, etc.; μclibc (http://www.uclibc.org) is a small version of the general C library; (micro window toolkits. . .).

Modern CPUs can decide to “optimize” your code, by changing the order of some statement. This means that reads and writes can be done in different orders, unless you take action to prevent it, such as a *memory barrier*. (This is a *hardware* barrier, which is different from the software barrier in Section 4.6.5!) Operating systems do these barriers for you, in a number of primitives, such as mutex, condition variable, or semaphore. The POSIX specification has more to say about this here (http://www.opengroup.org/onlinepubs/007904975/basedefs/xbd_chap04.html#tag_04_10).

Bibliography

URLs

KernelAnalysis-HOWTO (http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO.html). .

POSIX thread API concepts (http://as400bks.rochester.ibm.com/pubs/html/as400/v5r1/ic2924/index.htm?info/apis/rzah4mst.htm). .

Product Standard: Multi-Purpose Realtime Operating System (http://www.opengroup.org/branding/prodstds/x98rt.htm). .

The IEEE Computer Society Real-time Research Repository (http://cs-www.bu.edu/pub/ieee-rts/Home.html). .

Embedded Linux Howto (http://linux-embedded.org/howto/Embedded-Linux-Howto.html) , Sebastien Huet.

Linux for PowerPC Embedded Systems HOWTO (http://members.nbci.com/_XMCM/greyhams/linux/PowerPC-Embedded-HOWTO.html) , Graham Stoney.

Using Shared Memory in Real-Time Linux (). , Frederick M. Proctor.

The `Documentation` *directory of the Linux kernel* .

comp.realtime (news:comp.realtime) .

Real-time Linux mailinglist (/www.rtlinux.org) .

LinuxThread FAQ (http://linas.org/linux/threads-faq.html) .

comp.programming.threads FAQ (http://www.lambdacs.com/cpt/FAQ.html) .

Real-time FAQ (http://www.realtime-info.be/encyc/techno/publi/faq/rtfaq.htm) .

Real-time Linux FAQ (http://www.rtlinux.org/rtlinux.new/documents/faq.html). .

[Locke2002] *Priority Inheritance: The Real Story (http://www.linuxdevices.com/articles/AT5698775833.html)* , Doug Locke.

[Yodaiken2002] *Against priority inheritance (http://www.linuxdevices.com/files/misc/yodaiken-july02.pdf)* , Victor Yodaiken.

Linux, Real-Time Linux, & IPC (http://www.ddj.com/documents/s=897/ddj9911b/9911b.htm) , Frederick M. Proctor.

Linux Kernel Hacking HOWTO (http://netfilter.kernelnotes.org/kernel-hacking-HOWTO/kernel-hacking-HOWTO.html) , Paul Rusty Russell.

Linux Kernel Locking (http://netfilter.kernelnotes.org/unreliable-guides/kernel-locking/lklockingguide.html) , Paul Rusty Russell.

[Hyde97] *Interrupts on the Intel 80x86 (http://www.ladysharrow.ndirect.co.uk/library/Progamming/The%20Art%20of%20Assembly%20Language%20Progr* , Randall Hyde.

The ZEN of BDM (http://www.macraigor.com/zenofbdm.pdf) , Craig A. Haller.

MC68376 manuals (http://ebus.mot-sps.com/ProdCat/psp/0,1250,68376~M98645,00.html). , Motorola.

Articles and books

[Arcomano2002] *KernelAnalysis-HOWTO (http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO.html)* , Roberto Arcomano, 2002, The Linux Documentation Project.

[Barr99] *Programming embedded systems in C and C++*, Michael Barr, 1999, O'Reilly.

[BrinchHansen73] "Concurrent Programming Concepts", Per Brinch Hansen, 223–245, 5, 4, 1973, *ACM Computing Surveys*.

[BurnsWellings2001] *Real-time systems and Programming Languages*, 3, Alan Burns and Andy Wellings, 2001, Addison-Wesley.

[posa96] *Pattern-oriented software architecture: a system of patterns* , Frank Buschmann, Regine Meunier, and Hans Rohnert, 1996, Wiley Chicester.

[Dijkstra65] "Cooperating sequential processes", Edsger Wybe Dijkstra, 43–112, 1968, *Programming Languages*, Edited by F. Genuys, Academic Press.

[gof94] *Design Patterns Elements of Reusable Object-Oriented Software* , Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, 1994, Addison Wesley.

[Galli2001] *Distributed Operating Systems*, Doreen L. Galli, 2000, Springer.

[Gray78] "Notes on database operating systems", J. Gray, 394–481, 1978, *Operating systems: an advanced course*, Edited by R. Bayer, Edited by R. Graham, Edited by and G. Seegmuller, Springer.

[Herlihy91] "Wait free Synchronization", M. Herlihy, 124–149, 13, 1, 1991, *ACM Transactions on Programming Languages and Systems*.

[Herlihy93] "A Methodology for Implementing Highly Concurrent Data Objects", M. Herlihy, 745–77, 15, 5, 1993, *ACM Transactions on Programming Languages and Systems*.

[Hoare74] “Monitors, an operating system structuring concept”, C.A.R. Hoare, 549–557, 1974, 17, 10, *Communications of the ACM*.

[Johnson97] “Frameworks = (components + patterns)”, R. E. Johnson, 39–42, 40, 10, 1997, *Communications of the ACM*.

[LampsonRedell80] “Experiences with processes and monitors in Mesa”, Butler W. Lampson and Redell W. David, 105–117, 1980, 23, 2, *Communications of the ACM*.

[Mealy55] “A method for synthesizing sequential circuits”, G.-H. Mealy, 1045–1079, 1955, 34, 5, *Bell System Technical Journal*.

[Lewine91] *POSIX Programmer’s Guide: Writing Portable UNIX Programs*, Donald Lewine, 1991, O’Reilly.

[Nutt2000] *Operating systems : a modern perspective*, Gary J. Nutt, 2000, Addison-Wesley.

[Rubini2001] *Linux Device Drivers (http://www.oreilly.com/catalog/linuxdrive2/)*, 2, Alessandro Rubini and Jonathan Corbet, 2001, O’Reilly.

[Sakamura98] *μITRON 3.0: An Open and Portable Real-Time Operating System for Embedded Systems*, Ken Sakamura, 1998, IEEE Computer Society.

[Simon99] *An Embedded Software Primer*, David E. Simon, 1999, Addison-Wesley.

[Stevens99] *UNIX Network Programming. Interprocess Communications*, W. Richard Stevens, 1999, Prentice-Hall.

[Walmsley2000] *Multi-threaded programming in C++*, Mark Walmsley, 2000, Springer.