

Real-Time Linux Operating Systems

Peter Wurmsdobler

Centre de Transfert des Microtechniques
39, avenue de l'observatoire, 25000 Besançon, FRANCE

voice: +33.3.81.47.70.20 fax: +33.3.81.47.70.21

WWW: <http://www.ctm-france.com>

E-mail: peterw@ctm-france.com

19th February 2001

Contents

1	Introduction	4
2	Systems	4
2.1	Computer systems	4
2.1.1	General purpose computer	5
2.1.2	Software – hardware	5
2.1.3	System software – application software	5
2.1.4	User mode – supervisor mode	6
2.2	Software systems	6
2.2.1	Real time systems	6
2.2.2	Embedded systems	6
2.2.3	Organic systems	6
3	Operating systems	7
3.1	Basic mechanisms	7
3.1.1	Events	7
3.1.2	Exception	7
3.1.3	Interrupts	8
3.1.4	System calls	8
3.2	Process management	9
3.2.1	Processes	9
3.2.2	Threads	9
3.2.3	Task	9
3.2.4	Scheduler	9
3.2.5	Top and bottom half	10
3.3	Inter process communication	11
3.3.1	Signals and their handlers	11
3.3.2	Mutual Exclusion	12
3.3.3	Semaphore	12
3.3.4	Conditional variable	12
3.3.5	Barrier	12
3.3.6	Atomic operation	12

3.4	Consequences	12
3.4.1	Reentrant functions	13
3.4.2	Race condition	13
3.4.3	Deadlock	13
3.4.4	Priority inversion	13
4	Real time operating systems	13
4.1	Real time quantification	14
4.1.1	Response time or latency	14
4.1.2	Deadline	14
4.1.3	Jitter	14
4.2	Real time systems definitions	15
4.2.1	Hard real-time	15
4.2.2	Soft real-time	16
4.2.3	Firm real-time	16
4.2.4	Non-real-time	16
4.3	Real time scheduling	17
4.3.1	EDF-scheduler	17
4.3.2	RM-scheduler	17
4.4	Real time performance	17
4.5	Real time operating system implementations	17
4.5.1	Kernel replacement	18
4.5.2	Kernel modification	18
4.5.3	Kernel coexistence	19
5	Linux based real-time operating system	20
5.1	RTLinux	21
5.2	RTAI	22
5.3	Linux/RK	23
5.4	KURT	23
5.5	RED	23
5.6	Other implementations	24

1 Introduction

GNU/Linux [1] has become a powerful operating system challenging the operating systems and computer market, because it is open source and adaptable to different hardware and computing problems. As a general purpose operating system, however, the Linux kernel optimises average performance and is not that appropriate for real-time applications as such. For this reason there are some groups spread around the world developing modifications to the Linux kernel in order to provide a real-time operating system, all of them following different strategies. The resulting real-time operating systems are already used in many sophisticated real-time applications and form a good basis for embedded systems.

Before real-time Linux implementations are addressed, some general definitions should be introduced to form a minimum vocabulary for this text, followed by essential terms for operating systems. Both sections can be skipped for those familiar with computer and operating systems. The succeeding section will discuss basic notions of real-time operating systems, a section which can be skipped by readers familiar with terms like “hard real-time”. Finally, as the main part of this text, the most important implementations of real-time Linux are presented.

2 Systems

A system is a black box with internal states and bounds defining what belongs to it and what does not (Fig. 2). It interacts with its environment through inputs which can influence the system behaviour, and its outputs which are determined by the system, and which in turn can influence the environment. What is the relation to a computer and its operating system then ? The following will clear this point.

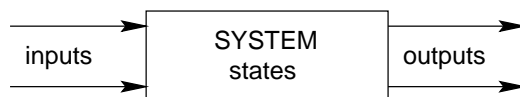


Figure 1: *A general view of a system, its inputs, outputs and states.*

2.1 Computer systems

A computer is a deterministic electro-mechanic machine for solving problems, hence a system in the aforementioned sense. For the purpose of problem solving, the computer system is able to accept and decode *instructions* which are actually macro-instructions implemented by microcode with microinstructions in the computer’s processor, but which form the computer specific *instruction set* as hardware abstraction layer. There are different computer systems, from micro-controllers up to mainframes, but in this paper we are talking about general purpose computers.

2.1.1 General purpose computer

The computer's processor or central processing unit (CPU) processes data according to such instructions stored in memory with the ability to feed back results as data and treat instructions as data and vice versa. This common type of computer, the so-called *general purpose computer*, permits the adaptation of one physical implementation to solve even different problems as long as they can be expressed in instruction sequences called *programs* which are stored in memory. For this reason a such computer sometimes is called *stored program computer*, too. According to the architecture proposed by Johann von Neumann in the forties, most general purpose computers adopt more or less sophisticated the architecture as depicted in Fig. 2.1.1.

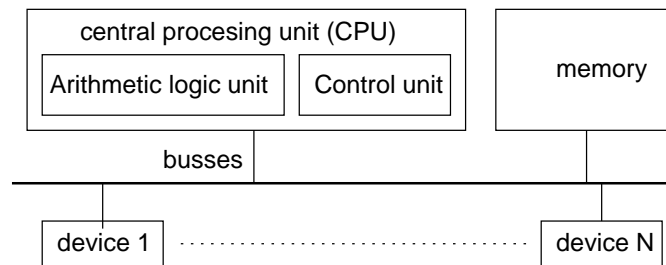


Figure 2: Von Neumann architecture with a central processing unit (CPU) including the arithmetic logic unit and the control unit, memory and io devices.

2.1.2 Software – hardware

The general purpose computer can logically be broken apart into two parts, hardware and software in a simple manner: the ensemble of programs (sequence of instructions) is called *software*, the rest is called *hardware* with its resources like processor(s), memory and i/o devices (following the classical von Neumann architecture). Software is the part of the general purpose computer changes to which alternate its behaviour. But note: one person's software is another person's hardware !

2.1.3 System software – application software

Software is in general divided into two categories, *system software* and *application software*. System software provides a general programming environment, e.g. from resource abstraction and management of the underlying hardware up to math and C libraries and even a windowing system. Application software does actually the useful work, it solves some problems like the computation of the meaning of life, the reason for which you started to deal with programming or to use a computer system. In-between are tools like compilers, assemblers, linkers, etc. but the same kind of distinction holds for application and system software as it does for hard and software: one person's application software is another person's system software, a basic rule for abstraction.

2.1.4 User mode – supervisor mode

Nowadays processors for general purpose computers know different modes of operation for executing instructions with more or less privileges for addressing memory or for instructions which can be executed. An Intel i86 architecture, for example, differs between 4 modes, two of them are used in Linux, one the so-called user space mode with limited access to memory and instructions, the other called supervisor mode with unlimited memory access and instructions.

2.2 Software systems

In a large sense a computer system or the software running on it, hence the software itself, represents a system: the input can be signals from the keyboard, digital signals on i/o ports or analogue signals on a DAQ board. The internal state is represented by all variables of the computer and the system behaviour is defined by its programs. The output of this software system can then be digital signals on i/o ports, an image on a monitor or analogue values on a DAQ board. There are different computer systems, from micro-controllers up to mainframes, but in this paper we are talking on general purpose computers.

2.2.1 Real time systems

The expression “real-time systems” denotes in general a software system in which timeliness is an integral part of the correctness of the systems output and behaviour. In a large sense any software system is real-time, but at different levels of acceptance for the user, of consequences for the system’s environment and of strictness for the word “timeliness”.

2.2.2 Embedded systems

A software system that is totally embedded by the hardware it controls is generally referred to as embedded system. These system span a wide range, but in general embedded systems are low memory systems and have restrictions with respect to available mass-storage devices as well as reduced peripheral devices. In former days special hardware was used mostly in conjunction with proprietary software for the implementation of embedded systems, nowadays, even general purpose computers or their components can be employed to form an embedded system.

2.2.3 Organic systems

A software system that does not depend bigly on the hardware it runs on and provides a generalised user interface.

3 Operating systems

The *operating system* (OS) is all system software underneath a certain abstraction level, used for resource management and device abstraction, especially if they are shared among more users and programs, basic system services like network connection and domain name resolution, printer and mail spooler, etc. The “atomic” part of the operating system is usually called *kernel* and encapsulates in general the most critical parts of the operating systems software. This trusted part of software is usually run in a privileged mode, the so-called kernel mode.

3.1 Basic mechanisms

Once any software is in execution on a computer, it would possibly never stop, because the CPU continues executing instructions one after each other following the fetch and execute cycle even for no operations. However there are some means to change the state of the processor and hence the flow of control, the so-called *events*.

3.1.1 Events

All CPUs are designed to stop on certain events, e.g. a division by zero or any other signal from a hardware device or some software. A boot up of the computer operating system, some bootstrap software installs a handler for each type of these events. This mechanism is vital for the entire computer system, in particular for a multi-processing system.

If an event should be handled in a specific way, a function or thread will be programmed that can respond to this event, e.g. update the pixels on the screen if the mouse moves. The association of this function or thread with a specific event makes it to the handler of this event. The handler will be called by the operating system on occurrence of the event, to do so the handler must be registered with the operating system.

Based on this event concept, the operating system’s kernel offers two interfaces, one to the application and systems software above, and another to the underlying hardware. Services of the operating system kernel for the application software can be accessed by use of system calls, hardware can be served by means of interrupts as depicted in Fig. 3.1.1 and explained in the following.

3.1.2 Exception

If something unforeseen happens like a division by zero triggered by some application software or a page fault triggered by the memory management unit, this exceptional situation is treated by the corresponding handler.

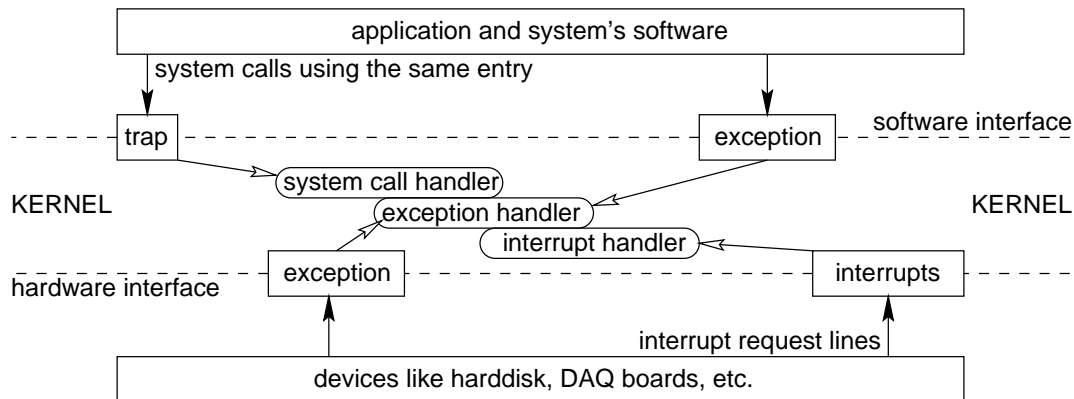


Figure 3: The operating systems kernel with its two interfaces, system calls and interrupts, respectively, and exception handling.

3.1.3 Interrupts

An *interrupt* is a signal that initiates an event, either as a hardware or a software interrupt. All processors have the capability to receive these signals via dedicated interrupt lines or registers. Devices in a computer, for example, can signal the CPU that they want to be served by an interrupt request. If an interrupt line is set by some device, the processor will halt execution of the current process, and jump to a its handler, the so-called interrupt service routine (ISR), a special event handler.

On normal processors, there are more interrupt lines with a certain priority among them. To determine which interrupts actually can reach the system a bit-mask is used to enable/disable interrupts. There is one interrupt called the non maskable interrupt (NMI) which is connected to the system clock, but it is treated as exception.

On asserting an hardware interrupt the system will call the associated interrupt service routine, the time from the assertion of the interrupt (the electric signal being active on the interrupt pin) to the point where this interrupt service routine is called is defined as the interrupt response time. In practice the interrupt response time is used for the time from asserting the interrupt until the system will acknowledge the interrupt or respond with a noticeable action, this time is therefore a little larger than the "theoretical" interrupt response time.

3.1.4 System calls

Interrupts come from hardware, *system calls* come from software. The set of system calls constitute the interface between user space and kernel space. Whenever a user space program requests a service from the operating system, it issues a system call, e.g. to open a file or to map memory in the processes space or to access some other hardware.

The system call itself is implemented by means of soft interrupts or so-called *traps* which notify the CPU to stop and to jump to the system call passed by the trap call. Here the

mode bit of the CPU is set to kernel mode and the application software enter kernel code. The kernel then checks a lot of things on behalf of the application software.

3.2 Process management

If there was only one program in execution at any time, only resource abstraction would be required from the operating system by some hardware drivers, the different programs can rely on and build upon, e.g. drivers for hard disks or data acquisition boards. However, in a multi-programming environment, resources have to be shared which makes the introduction of additional operating system components for their management necessary.

3.2.1 Processes

A process can be viewed as a program in execution, i.e. the program itself and its context. Hence, a process includes the object program, i.e. the program text the data the program acts on, the resources required by the program and the status of its execution. The operating system will keep track of all processes running in a multi-process environment by attaching a process descriptor for each. The process constitutes the fundamental unit of computation in a computer system and is managed by the operating system or more precisely by its kernel.

Each process has a “life-cycle” from create to runnable to running and finally to terminated. From the running state a thread can go into a sleep or stopped state from which it will return to a runnable state before changing state again. A process can switch between kernel and user mode in the running state only.

3.2.2 Threads

Sometimes a thread is called a lightweight process, as it is a unit of computation with a minimum set of context informations, its own program and its state. A thread is associated with a process and its resources, especially with the program text, but defines an independent flow of control within the process. A process that has more than one flow of control (threads) is then called multi-threaded.

3.2.3 Task

In the process model a task represents a process, in a thread model a task can be a process (single threaded process) or a thread within a multi-threaded process. In the following the term task is used interchangeably for thread or process as computational unit.

3.2.4 Scheduler

The operating system must provide a possibility to switch between processes or more general between computation entities like processes and threads. There must be some “meta process” responsible for this process switching or the *scheduling* of a set of computation units. The

scheduler, some piece of software within the kernel called periodically or on special occasions accomplishes this task. It preempts a running process, rushes through the task list searching for a runnable process, selects another one and switches to it according to the processes priority and to some policy, called the scheduling policy, like Round Robin or First In First Out.

First In First Out (FIFO) scheduler means that all processes/threads at the same priority level are scheduled in the order they arrived a queue maintained by the kernel. When the scheduler is called the highest priority is checked first, if there is no thread runnable in the highest priority level the next level is checked etc. A job scheduled with a policy of FIFO can monopolise the CPU, there is no mechanism to preempt a job running with FIFO set.

Round Robin (RR) scheduling (RR), again there are different priority levels available, but each schedulable entity has a defined time-slice. If it does not exit or block before the time-slice expired it will be preempted by the kernel and the next runnable thread is scheduled.

Every schedulable entity will be called by the scheduler to execute in a order specified by its priority level. POSIX specifies a minimum of 32 priority levels. Besides the priority of a task its scheduling policy (FIFO,RR) will influence when it is run as well. A priority level only specifies its rank within the scheduling policy, that is when it is run in relation to other tasks in that same scheduling class.

Whenever one process is preempted and another one is loaded, a so-called context switch happens: the context of the process is saved in some place and the currently running thread is removed from the processor. The context of a different thread (or processes) is then loaded into the CPU registers and its execution is started.

Up to now, there was no explication on how the scheduler comes to run. One possibility would be that cooperative processes yield the CPU voluntarily after some time or after an exit statement. In most operating systems, however, no process is trusted and involuntary CPU sharing is provided on various occasions. On the one hand the scheduler is started by the system timer interrupt handler at a fixed frequency, under Linux usually 100Hz. On the other hand, each system call will conclude with a scheduler call. In both cases a switch to another process can happen.

3.2.5 Top and bottom half

Any hardware device is usually controlled by some system software called driver. Since most devices are slow in comparison to the CPU, a dual concept is used when communicating with such devices. First, if a user process wants to access a device it will issue a system call to the kernel by some read operation. Some driver code attached to it and referred to as *top half* will then program the device to do something, e.g. to get an analogue value from a DAQ board. After this action, the *top half* driver will notify the scheduler that an io operation has has been started on behalf of the respective process. Hence the scheduler will schedule another process and put the latter on a wait queue with the result that this one is “blocked”.

Given that the driver for the mentioned device had registered a handler for the device’s interrupt, the handler will carry out the most urgent instructions to communicate with the device, e.g. notify to the device that it will served. Then he will probably re-enable interrupts.

The more laborous part of the device communication will then be deferred to a part called *bottom half* since without this technique the CPU would be blocked for a too long period. It is the scheduler who will perform the rest of the work of all bottom halves whenever he is called and before he returns to the user space. After finishing these bottom halves, notably the process who triggered the respective io operation will be removed from the wait queue, become runnable and can be scheduled again. Here the user code will continue where it had been blocked.

3.3 Inter process communication

In the previous parts processes were independent units of computation. However, if multiple processes or threads work together by some means, some principles of cooperation have to be respected, or if different processes rely mutually on results of one each other, they have to be synchronised. The most common means of inter process communication are shared memory which is accessed by more than one process, signals, semaphores and mutexes.

3.3.1 Signals and their handlers

A signal is a numeric value delivered to a process via system call, describing an action to be taken by the process. The process may accept a signal or mask it. If a process has a signal handler installed for the signal number sent this handler will be executed on arrival of the signal. Signals issued from a thread within a process can be posted to a specific thread (via the thread id), signals sent between processes are received at the process level and are not directed to a specific thread.

All signals that reach a thread from an outer source, that is a different thread is posting a signal via special command. Not all thread functions are async safe. An asynchronous signal is delivered to the process and not to a specific thread within a multi-threaded process.

A thread function that can handle asynchronous signals without leading to race conditions or synchronisation problems (like blocking other threads infinitely, or leading to inconsistency in global variables) are considered async safe functions. Functions that are not async safe should be used with these possible sideeffects in mind, that is the capabilities state should be set appropriately.

To manage asynchronous signal on a process level signal handlers are installed. These can then be called by the thread that received the signal to respond. Signal handlers are installed on the process level and not at the thread level, if a async signal is received it is not said which thread of the process will handle it. Only signals issued from within the process can be sent to a specific thread thread IDs are only unique within a process .

A synchronous signal is any signal that is the result of the threads action. An example of a synchronous signal would be a thread that does a division by zero causing an exception. Synchronous signals are delivered to the thread that caused the operating system to post it and not to the thread.

3.3.2 Mutual Exclusion

A Mutex (Mutual Exclusion Object) is an object that allows multiple threads to synchronise access to shared resources. It has two states: locked and unlocked. Once a Mutex has been locked by a thread all other threads that try to access it will block until the thread that acquired the mutex unlocks it, after this one of the blocked threads will acquire it.

3.3.3 Semaphore

The simplest form of a semaphore (invented by the Dutch computer scientist Dijkstra and means a red light at a junction of two streets) is equivalent to a mutex, the binary semaphore. Associated with a semaphore is a counter that defines the number of threads that can access a protected resource via the semaphore. On access of the protected resource a thread acquires the semaphore by decrementing the counter, if the counter reaches 0 no further thread can access the protected resource. When a thread releases the protected resource it increments the semaphore again.

3.3.4 Conditional variable

Conditional variables are a complex synchronisation mechanism comprised of a conditional variable and its predicate as well as an associated mutex. A thread will acquire the mutex and then wait until the condition required to continue is signalled, then performs the task depending on the condition and finally release the mutex.

3.3.5 Barrier

A barrier is a thread synchronisation primitive based on conditional variables. It is a point in the execution stream at which a set of threads will wait until all threads requiring synchronisation have reached it, after all thread have reached the barrier the condition predicate is set TRUE and execution of all thread can continue.

3.3.6 Atomic operation

In order to guarantee the functioning of the aforementioned communication principles, a core mechanism is necessary, the atomic operation. An atomic operation is an execution operation during which a context switch can occur but state is preserved, during atomic operations it is legal to assume that conditional variables, mutexes, etc. will be unchanged. An atomic operation behaves as if it were completed as a single instruction.

3.4 Consequences

Based upon the definitions up to now, the notion of some terms is vital to the understanding of a multiprocess or multi-threaded system.

3.4.1 Reentrant functions

A reentrant function will behave identical if called by multiple threads at the same time as it would if it is only called once, that is any synchronisation or access of global data is handled in a way that it is safe to call these functions multiple times, and have them run interleaved.

3.4.2 Race condition

If two executing entities compete for a resource and there is no control provided as two when the resource is accessed safely, unpredictable behaviour can occur. Race conditions can occur with any shared resources if no appropriate synchronisation is done by all entities that require access to this resource.

3.4.3 Deadlock

If synchronisation primitives are used inconsistently then two threads may cause a deadlock (interactive deadlock). An example of such a setup is two thread that each block on a mutex that the other thread acquired, since both threads are blocked non of them will free the mutex they hold and thus both are blocked infinitely.

3.4.4 Priority inversion

In operating systems technology the term “priority inversion” is known for two different phenomenon:

1. In a multitasking system that supports task priority assignment high priority tasks would always be run and a low priority task would never get hold of the CPU, to prevent this the priority of a job is modified after it is run, so that it will not stay at the top all the time, this means that a low priority job sooner or later has the highest priority and actually gets run. This priority inversion is what guarantees that every job get run at some point even in a heavily loaded system.
2. If a high priority job waits on a low priority job by some means of thread synchronisation (mutex,condvars,etc.) then the high priority job is reversed to a de-facto low-priority job.

4 Real time operating systems

Generally speaking, general purpose operating systems with its mechanisms and principles as described above ensure the logically correct cooperation of processes and the programs executed in them. However, the timeliness of any process and its result does not matter. An operating system taking also in account timing constraints in its mechanisms when scheduling processes, is called a real-time operating system, i.e. a process has to be scheduled for sure

at a certain time. Thus, the correctness of the system is a question of both the correctness of the output and its timeliness.

4.1 Real time quantification

The common sense approach to “real-time” is that these kind of systems have to be very fast, in particular compared to the human sensitive system. Real time systems are sometimes fast, but this is not a sufficient condition. In order to quantify the term “real-time” some expressions are defined.

4.1.1 Response time or latency

A computer and software system never responds instantaneously, but after some time. For example, when using a word processor on a general purpose computer, the graphical representation of a character is displayed some milliseconds after the key has been hit on the keyboard. More general, the time delay between system input and system output is called *response time* or *latency*.

4.1.2 Deadline

A time constraint for a process is called deadline, i.e. this is the time when a result has to be delivered. Commonly *a priori* deadlines as for processes which can be scheduled in advance, and *sporadic* or *instantaneous* deadlines such as interrupts with their service routines, are distinguished.

4.1.3 Jitter

The term jitter is used for describing the uncertainty of an event in terms of time. An event, for example, is supposed to be scheduled at a predefined sequence of time instances T_1, T_2, \dots, T_n which constitute deadlines, but actually there is a certain time deviation at each such instance like $\Delta T_1, \Delta T_2, \dots, \Delta T_n$. This phenomenon of time deviation generally is called jitter and is depicted in Fig. 4.1.3.

Sometimes you can read “the jitter is 1ms” which is a quantification of the aforementioned phenomenon and applies in statistical terms for the mean value of a sequence of time deviations, i.e. $\text{jitter} = \text{mean}(\Delta T_i)$. In other places, however, one can read expressions like “the maximum jitter is 1ms” which signifies the maximum time deviation of an event or a sequence, i.e. $\text{jitter} = \text{max}(\Delta T_i)$. So the word “jitter” is used for a phenomenon, statistically for a mean value or for a single event.

If the term jitter is applied to scheduling, the expression “scheduling jitter” in general means the mean value for the sequence of time deviations for each time instance a process or thread ought to be started or resumed. The term “maximum scheduling jitter” applies though to the maximum time deviation for a process or thread to be scheduled.

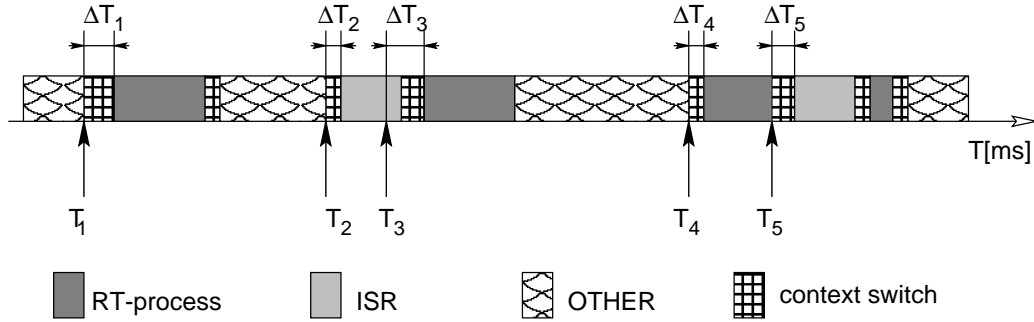


Figure 4: *Scheduling jitter sequence for one real-time process and one interrupt service routine (ISR).*

4.2 Real time systems definitions

Most systems are “real-time” in a large sense, but they may differ in the consequences of being not in time and in the strictness of the specification. If a word processor does not respond within a time period significant for a human being, its use is exhausting, if a failure procedure of a nuclear power plant is not called on urgent demand, the result is a disaster. Different levels of timeliness are distinguished in statistical terms with the area of applications they are good for as follows.

4.2.1 Hard real-time

An operating system is considered to be hard real-time *iff* all deadlines are strictly met and all scheduling deviations are within a specified limit value, i.e. $\max(\Delta T_i) < \Delta T_{limit}$. For example, a process is scheduled within a specified tolerance of 1ms any time when it ought to, and the interrupt response time for any interrupt issued by a device is less than some specified $100\mu s$, too. The consequent properties are:

- system time is a managed resource,
- the worst-case scheduling jitter is guaranteed,
- the maximum interrupt response time is guaranteed,
- no realtime event is ever missed,
- the system response is load-independent,
- the system is event deterministic.

From the application point of view, this system property is required for safety critical applications where missing a deadline has catastrophic consequences. In a digital control system where the control thread has to be scheduled with a certain period, missing a deadline can make the control loop unstable.

4.2.2 Soft real-time

An operating system is considered to be soft real-time iff all deadlines are met in a *statistical* sense, the mean value of schedule time deviation is less than a predefined tolerance, i.e. $\text{mean}(\Delta T_i) < \Delta T_{mean}$. For example, a process is scheduled within a mean time deviation of 1ms, i.e. it may happen, that sometimes the scheduling delay is bigger and some deadlines are lost. The consequent properties are:

- the system can guarantee worst-case average jitter,
- the average interrupt response time will not exceed a maximum value,
- events may be missed occasionally,
- the system is statistically deterministic.

From the application point of view, this OS property is used for example in multimedia applications where missing a deadline has no catastrophic consequences. If a video frame is lost due to a missed deadline, no harm is done to anybody, i.e. the correctness of the application is invariant to missing of deadlines.

4.2.3 Firm real-time

Firm real-time is a mixture of soft realtime and hard realtime with the statistical approach of soft real-time concerning the mean value of jitter, but adding the requirement that the jitter variance must be beyond a certain value, too, e.g. $\text{mean}(\Delta T_i) < \Delta T_{mean}$ and $\text{variance}(\Delta T_i) < \text{variance}_{\Delta T}$.

4.2.4 Non-real-time

“Non-realtime” systems are the systems most often used. These systems are simpler and are able to utilise optimisation strategies that are contradictory to realtime requirements, for example caching and buffering in order to improve average performance and throughput. Non-realtime systems are characterised by:

- no guaranteed worst-case scheduling jitter at all,
- no theoretical limit on interrupt response times,
- no guarantee that an event will be handled,
- the system response is strongly load-dependent,
- the system timing is a unmanaged resource.

Non-realtime systems are unpredictable even at a statistical level. System reaction is highly dependent on system load. Non-realtime systems can use optimisation strategies that are unsuited for hard or soft realtime systems.

4.3 Real time scheduling

As long as there is only one task or thread in a real-time system, no real-time operating system would be necessary, e.g. a single control loop running on a processor. However, usually more threads are involved in a real-time system, like several control loops at different sampling times and with different priorities, and a watchdog at a low frequency. Given that each thread can be preempted, different policies for scheduling multiple real-time threads are commonly known as follows.

4.3.1 EDF-scheduler

In the earliest deadline first (EDF) scheduler it is not the priority of a task which is used to decide what task to run, rather simply the task with the closest deadline, that is the least time left until it should be run is taken as scheduling criteria. This scheduling strategies has a "flat" priority and is optimal for systems that have to handle asynchronous events and non-periodic realtime tasks.

4.3.2 RM-scheduler

The task priority is inverse proportional to the task's period given that there is a common divisor for the period. The criteria is that all tasks meet the requirement

$$\frac{C_1}{T_1} \times \frac{C_2}{T_2} \times \dots \times \frac{C_n}{T_n} < n \cdot \left(2^{\frac{1}{n}} - 1\right)$$

with C_n being the works case execution time and T_n the period of the respective task.

4.4 Real time performance

Hard realtime systems will generally have slightly lower average performance than soft realtime systems, which in turn are generally not as efficient with resources as non-realtime systems. On the other hand, non-realtime systems are not at all predictable and soft-realtime systems are only statistically predictable. Only hard realtime systems are deterministic with respect to high-priority tasks. From the above definitions, it is clear that the border between non- and soft-realtime is difficult to define precisely. In general, these definitions will vary depending on the criteria that are emphasised when describing such a system.

4.5 Real time operating system implementations

The fundamental problem of an RTOS is that users have conflicting demands with respect to system design. On the one hand, an RTOS should obviously be capable of realtime operations. On the other hand, users want access to the same rich feature sets found in general-purpose operating systems which run on desktop PCs and workstations. How can one make an operating system to be real-time. Several approaches exist as shown in the following.

4.5.1 Kernel replacement

The first approach is to replace an existing non-real-time kernel by a small real-time kernel providing the same API. The design guidelines for an RTOS include the following: It needs to be compact, predictable and efficient; it should not need to manage an excessive number of resources; and it should not be dependent on any dynamically allocated resources. However, if one expands a small compact RTOS to incorporate the features of typical desktop systems, it is hard (if not impossible) to fulfil the demands of the core RTOS. The pro and cons are:

- + Complete new design of the kernel adapted for the real-time problem at hand and optimised for real-time.
- + Allows to grow and evolve with the market by still keeping the same API, in particular in the embedded field.
- The OS becomes very complex. This makes it difficult to ensure determinism, since ALL core capabilities must be fully preemptive.
- Drivers for hardware become very complex. Since priority inversion must not occur, drivers must be able to handle situations in which they are not being serviced.
- As complexity increases, dependencies become very complex. This makes systems hard to analyse and debug.
- Since the core system is an RTOS, the vast amount of free software that is available cannot (in most cases) be used unmodified. It is even harder to use unmodified commercial software (where source code is not available), because it is almost impossible to determine interactions between the software and the RTOS.
- Many mechanisms for efficiency, like caching and queueing, become problematic. This prohibits usage of many typical optimisation strategies for the non-realtime applications in the system.
- Maintenance costs of such a system are considerable for both developers and customers. Since every component of the system can influence the entire system's behaviour, it is very hard to evaluate updates and modifications with respect to the realtime behaviour.

4.5.2 Kernel modification

The most seemingly natural alternative strategy would be to add RT capabilities to a general purpose OS by modifying the kernel. One common technique is to insert preemption points into the kernel wherever it is safe to perform a context switch. Then the kernel checks again if a high priority process is runnable. The pro and cons are:

- + Performance improvement with little changes may be sufficient.
- + All general tools for debugging and development can be used.

- + Slight modification keeps the kernel near to the development thread.
- + Kernel changes are in the scheduler and bottom half treatment which is far away from the interrupt hardware.
- General purpose operating systems are event-driven, not time-triggered.
- General Purpose OS's are not fully preemptive systems. Making them fully preemptive requires modifications to all hardware drivers and to all resource handling code.
- Lack of built-in high-resolution timing functions entail substantial system modification.
- Modifying applications to be preemptive is very costly and error-prone.
- The use of modified applications would also greatly increase maintenance costs.
- Optimisation strategies used in general purpose OSes can contradict the RT requirements. For example, removing all caching and queueing from an OS would substantially degrade performance in areas where there are no realtime demands.
- Because such systems are very complex (and often not well-documented), it is extremely difficult to reliably achieve full preemption in such a system.

General purpose operating systems are efficient with resources. Because they don't manage time as an explicit resource, however, trying to modify such a system boils down to using it in a manner in which it was never intended to be used.

4.5.3 Kernel coexistence

The kernel is split into two parts, one part that runs as a general purpose OS with no hard realtime capabilities, and a second part that is designed around these realtime capabilities. This idea to run a full feature OS on top of a real-time executive is already often used, e.g. to add real-time capabilities to the Microsoft Windows operating systems. The pro and cons are:

- + Reduces all general features to a bare minimum.
- + Allows the non-realtime side of the OS to provide all the goodies that desktop users are used to.
- + Realtime side can be kept small, compact, fast and deterministic.
- Real time threads are executed in supervisor mode and no protection model is applied.
- Debugging in the kernel is difficult.
- API is mostly proprietary and limited.
- Application must be divided in a real-time and non real-time part which makes porting more difficult.

- Everything that is needed in real-time has to be duplicated, real-time networking, drivers, etc.
- Third party software has to be adapted and is not applicable in its native version.

5 Linux based real-time operating system

GNU/Linux is an operating system for a general purpose computer optimised for maximum throughput and average performance of each process. For this reason some problems have to be addressed like:

Coarse-grained synchronisation The Linux kernel uses coarse-grained synchronisation, which allows a kernel task exclusive access to some data for long periods. This delays the execution of any POSIX real-time task that needs access to that same data.

Kernel preemption Of course all user processes are preemptible by the kernel, in particular by the scheduler after a blocking system call. If the user process starts a kernel activity on his behalf, the kernel itself cannot be preempted any more (except for interrupt service routines). Only when the mentioned kernel activity is finished, a higher priority process can run.

Scheduler Meeting hard deadlines has not been foreseen in the Linux kernel as the scheduler and other kernel resources were designed for average performance. This effect is aggravated by the fact that the scheduler is called periodically and after each system call which in statistical terms increases its execution time proportional to the number of processes and reduces response time.

Interrupt disable In order to implement atomic operations and critical regions, the kernel will disable interrupts sometimes for longer periods, especially in device drivers. During this period which can take some milliseconds, the system does not respond to any other interrupt or signal.

Bottom halves In order to reduce interrupt response times by the time an interrupt service routine stays in its code most work is deferred to bottom halves. All bottom halves are executed before the scheduler can switch to a high priority user process, even if the bottom halves are not very important.

Task queueing Usually if a process blocks on a resource, it is put on a wait queue without order of priority. Once the resource is available, all these processes are made runnable and are hence schedulable with not necessarily the highest priority process winning the race.

Thread support The Linux kernel itself does not directly support threads. Rather the POSIX pthread library maps threads to classical UNIX processes which are slower to be treated.

Priority inversion Whenever multiple processes with different priorities compete for a resource, priority inversion can happen with a high priority process being blocked by a lower priority one holding a resource, but not schedulable. There are solutions to those problems but this is not addresses yet in the current Linux kernel.

Due to the fact that the Linux kernel is open source, however, it can be adapted and despite the difficulties and problems mentioned before and there are several real-time Linux implementations [2], either following the kernel replacement, kernel tuning or kernel co-existence way. The most prominent ones will be explained in the following.

5.1 RTLinux

The core idea behind RTLinux [4] is to run a full featured operating system as one thread of a real-time executive, so it follows the kernel co-existence concept. In a logical sense, “tasks” and “interrupts” are separated in two classes, the realtime ones running directly on behalf of the executive, and the non-realtime ones being executed within the common operating system, with some means of communication between these two priority spaces. “realtime” in this context is meant to be *hard realtime* as its deadlines have to be met in any case. In order to meet this goal, the general rule is to keep as many services in non-realtime Linux as possible if they are not inherently hard realtime, like data display or storage.

As general purpose operating system Linux optimises *average* performance, but is not appropriate for hard realtime. The RTLinux modifications to the kernel, however, put a thin layer underneath the Linux operating system. This core hard realtime mechanism takes over control of the low level interrupt handling from Linux: in some cases Linux kernel code is modified to make this takeover work smoothly. RTLinux applies most changes directly to the kernel source files, resulting in modifications and additions to numerous Linux kernel source files. The low latency interrupt handlers cannot be preempted by Linux acting then only on emulated interrupts. All other functionalities and communication means as schedulers, timers, POSIX-io, FIFOs, shared memory, semaphores and mutexes can be added by inserting a kernel module at runtime.

With the core functionality realtime ISRs can be installed already as simple real-time “application”. Once the modules for time and scheduling are inserted into the kernel, both “task” and “interrupt” priority spaces are created. From now on the thin hard realtime layer constitutes a single process per CPU running on the bare computer hardware with Linux being in the following just a low priority thread among realtime threads and interrupt service routines (ISRs). Linux as “task” works as usual in its own process space as illustrated by Fig. 5.1.

Form the user’s point of view, the computer can be used as full featured workstation in normal mode. In case of a realtime event, however, Linux is completely preempted and the realtime threads or interrupt service routines are treated immediately. Once they have finished and there is nothing else to do in the “hard realtime world”, Linux is scheduled again.

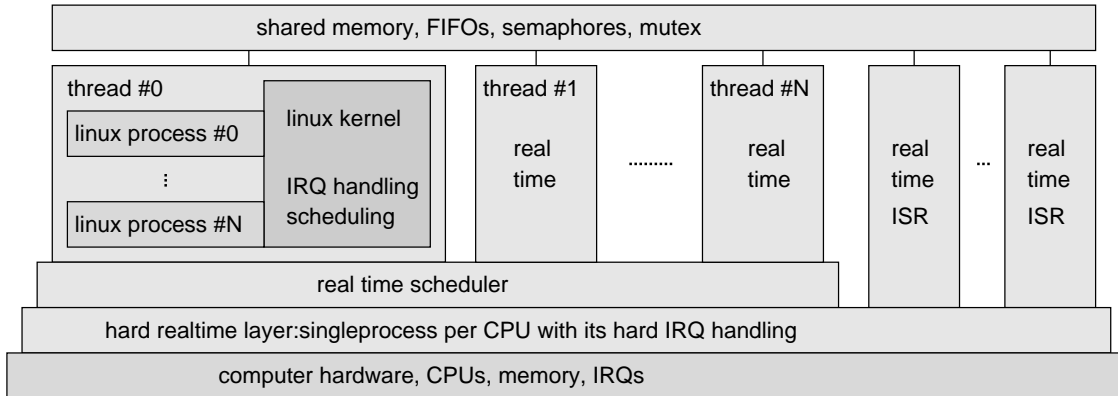


Figure 5: *Principle structure of RTLinux realtime implementation.*

5.2 RTAI

The architecture for RTAI [5] is quite similar to RTLinux and as the GNU/Linux operating system is run as the lowest priority task of a small real-time executive. The primary architectural difference between the two implementations is in how the real-time features are added to the Linux kernel. RTAI limits the changes to the standard Linux kernel by adding a hardware abstraction layer (HAL) comprised of a structure of pointers to the interrupt vectors, and the interrupt enable/disable functions. The HAL is implemented by modifying fewer than 20 lines of existing code, and by adding about 50 lines of new code. This approach minimises the intrusion on the standard Linux kernel and localises the interrupt handling and emulation code, which is a far more elegant approach.

Another advantage of the HAL technique is that it is possible to revert Linux to standard operation by changing the pointers in the RTHAL structure back to the original ones. This has proven quite useful when real-time operation is inactive or when trying to isolate obscure bugs. The HAL's impact on the kernel's performance is negligible, reflecting highly on the maturity and design of the Linux kernel and on those who contributed to its development.

RTAI's task scheduler allows hard real-time, fully preemptive scheduling based on a fixed-priority scheme. All schedules can be managed by timing functions and real-time events such as semaphore acquisition, clocks and timing functions, asynchronous event handlers, and include inter-task synchronisation.

RTAI provides efficient and immediate access to the hardware by allowing, if one chooses, interaction directly with the low-level PC hardware, without first passing through the interrupt management layers of the standard Linux kernel. The ability to individually assign specific IRQs to specific CPUs, as described in further detail below, allows immediate, responsive and guaranteed interface times to the hardware.

There is an additional feature: Linux-RT (LXRT). Since real-time Linux tasks are implemented as loadable modules, they are, for all practical purposes, an integral part of the kernel. As such, these tasks are not bounded by the memory protection services of Linux, and they have the ability to overwrite system-critical areas of memory, bringing the system

to an untimely halt. This limitation has been a large frustration to those of us who have erred during real-time task development.

This information has partially been taken from [11].

5.3 Linux/RK

Linux/RK [6] stands for Linux/Resource Kernel, which incorporates real-time extensions to the Linux kernel to support the abstractions of a resource kernel. A resource kernel is a real-time kernel (operating system) that provides timely, guaranteed and enforced access to system resources for applications. The Portable Resource Kernel has been integrated with Linux successfully. Linux/RK is said to be a soft real-time operating system.

This information has partially been taken from [6].

5.4 KURT

The KURT [7] patch builds on top of UTIME, which implements on demand microsecond resolution timers, adding a number of features. KURT enables the system to switch between normal mode, in which it functions as a normal Linux system, and real-time mode, in which it executes only designated real-time processes according to an explicit schedule. When in real-time mode, real-time processes can still access any of the system services that are normally available to non-real-time processes. While many approaches to scheduling can be easily implemented, KURT employs explicit scheduling, as it was found to be the most appropriate for our current set of firm real-time applications. Finally, KURT is firm real-time.

This information has partially been taken from [7].

5.5 RED

RED-Linux [8] provides a general scheduling framework to support different real-time scheduling paradigm in one operating system kernel. The scheduling paradigms supported including: priority-driven, time-driven and share-driven. It has a high resolution (microsecond) timer so that systems can have a very precise control on the time when a job must be executed. The scheduler implemented in the RED-Linux kernel is divided into two components: the Allocator and the Dispatcher. The Dispatcher implements the basic scheduling mechanism, and the Allocator implement the policy that manages the CPU time and system resources to meet the real-time constraints of user jobs. In this way, the scheduling policy may be easily modified without making changes on the low-level scheduling mechanism. The structure also allows well-designed application schedulers to be reused by other applications. We will collect the schedulers contributed by various projects implemented for different applications in a Allocator library to allow easy sharing and reuse.

This information has partially been taken from [8].

5.6 Other implementations

Due to the fact that the Linux kernel code is open source, there are a lot of other mostly soft real-time implementations with information partially taken from [2], among them:

SMART-Linux is adaptive kernel capable of reacting to system load and adapting real-time behaviour to guarantee quality of service. It was developed by Professor Dilma Silva at the University of Sao Paulo, Brazil.

ART Linux is a Real-Time extension to Linux, developed by Youichi Ishiwata at ETL and is inspired by RT-Linux.

Linux-SRT an extension to the Linux kernel which improves the performance of “soft real-time” applications such as in multimedia.

There are a lot of web sites dedicated to real-time Linux. The most complete collections are the Real-Time Linux Foundation [2] and Linuxdevices.com [11].

References

- [1] The Free Software Foundation, www.gnu.org.
- [2] The Real Time Linux Foundation, www.realtimelinuxfoundation.org.
- [3] Feiguin, Adrian E. (1999). Gtkplot, a high quality scientific plot widget, <http://www.ifir.edu.ar/grupos/gtk/>.
- [4] Yodaiken, Victor (1999). RTLinux, real-time Linux, FMSLab, Socorro, New Mexico, USA, <http://www.rtlinux.org/>.
- [5] Mantegazza, Paolo (1999). RTAI, real-time application interface, Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, Italy, <http://www.aero.polimi.it/>.
- [6] Rajkumar, Raj (2000). Linux/RK the Linux resource kernel. Carnegie Mellon University, USA <http://www.cs.cmu.edu/~rajkumar/linux-rk.html>.
- [7] Niehaus, Doug (1999). KURT, the Kansas university real-time Linux. University of Kansas, USA. <http://www.ittc.ukans.edu/kurt/>.
- [8] Lin, Kwei-Jay, (1999). REDLinux, Real-time and Embedded Linux. .
- [9] Motylewski, Tomasz (1999). mbuff, a kernel shared memory driver <http://crds.chemie.unibas.ch/>.
- [10] Peter Mattis, Spencer Kimball, Josh MacDonald (1999). Gtk+, the GIMP ToolKit, <http://www.gtk.org/>.
- [11] Rick Lehrbaum (1999). Linuxdevices.com, <http://www.linuxdevices.com/>.