The Abdus Salam International Centre for Theoretical Physics

United Nations Educational, Scientific and Cultural Organization

IAEA International Atomic Energy Agency

2057-12

First Workshop on Open Source and Internet Technology for Scientific Environment: with case studies from Environmental Monitoring

7 - 25 September 2009

REALTIME OPERATING SYSTEMS

Olexiy Tykhomyrov

Department of Experimental Physics
Dnepropetrovsk National University
Proulok Naukovij, 13
49050 GSP 50 Dnepropetrovsk
Ukraine

Contents

1. What is a Real-Time Operating System
2. The features of a Real-Time System
3. Design philosophies
4. RTOSes available on the market
5. Useful URLs

Real-Time OS: What it means (1)

- Look for information? Ask WIKIPEDIA!
- Here you are the first paragraph from its article about real-time operating systems:

  A Real-Time Operating System (RTOS) is a **multi-tasking** *operating system intended for real-time applications. Such applications include embedded systems (programmable thermostats, household appliance controllers), industrial robots, spacecraft, industrial control (see* **SCADA***), and scientific research equipment.*

- Assuming you do not know anything of RTOS. Is the information clear and understandable for you?
- I guess **no** due to some more terms you possibly did not meet before, and links to other definitions.

Real-Time OS: What it means (2)

- Not to use WIKIPEDIA?
- Better lets start from the very simple, not formal, but more clear definition:

  If a system responds to an external stimulus within a short specified time, we can name it the **Real-Time Operating System**.
- This definition, however, covers a very large range of systems, much wider, than we are usually going to deal with.

Real-Time OS: 'Human' RT Example

A data-base management system can justly claim to operate in real-time, if the operator receives replies to his queries within a few seconds. As soon as the operator would have to wait for a reply for more than, say, 3 seconds, she would get annoyed by the slow response and maybe she would object to the adjective ‘real-time‘ being used for the system. Apart from having unhappy users, such a slow data-base query system would still be considered a real-time system.

Real-Time OS: Assembly Line

The real-time systems we want to deal with are much more strict in requiring short response times than a human operator is.

Real-Time OS: Assembly Line

The real-time systems we want to deal with are much more strict in requiring short response times than a human operator is.

An operating system might be designed to ensure that a certain object was available for a machine on an assembly line. Such machines are called **robots**

Real-Time OS: Assembly Line

The real-time systems we want to deal with are much more strict in requiring short response times than a human operator is.

An operating system might be designed to ensure that a certain object was available for a machine on an assembly line. Such machines are called **robots**

Response times well below a second are usually asked for, and often a delay of a few milliseconds is already unacceptable. In very critical applications the response may even have to arrive in a few tens of microseconds.

Soft Real-Time Systems

To make the previous definition more accurate, in order to claim rightly that we are really having a real-time system, *we must specify the response time of the system*.

Soft Real-Time Systems

To make the previous definition more accurate, in order to claim rightly that we are really having a real-time system, *we must specify the response time of the system*.
If this response time can be occasionally exceeded, without any real harm being done, we are dealing with a **soft** real-time system.

Soft Real-Time Systems

To make the previous definition more accurate, in order to claim rightly that we are really having a real-time system, *we must specify the response time of the system*.
If this response time can be occasionally exceeded, without any real harm being done, we are dealing with a **soft** real-time system.
On the contrary, if it is considered to be a failure when the system does not respond within the specified time, we are having a hard real-time system.

Soft Real-Time Systems

Returning to our examples, a data-base query system will generally fall in the first category: it will make little difference if a human operator will have to wait occasionally 4 seconds instead of usual 3 seconds, and nobody will dare to speak of a failure, as long as the replies to the queries are correct.
For our second example, an assembly line, if an object is not available (or available at the designated time), the robot just would wait and then continue to work, but the production output might be lower.
We are still talking about the RTOS, but call them the soft Real-Time Operating System.

Hard Real-Time Operating System

If it is considered to be a failure when the system does not respond within the specified time, we are having a **hard** real-time system. In a hard real-time system, exceeding the specified response time may well result in serious damage of one sort or another, or in extreme cases even in the loss of a human life. The computation results obtained after that specified time have either a zero-level of usefulness or have a high rate of depreciation as time moves further from that event before the system gives a response. Such parameter sometimes called *real-time system deadline*.

Hard Real-Time Operating System

If it is considered to be a failure when the system does not respond within the specified time, we are having a **hard** real-time system. In a hard real-time system, exceeding the specified response time may well result in serious damage of one sort or another, or in extreme cases even in the loss of a human life. The computation results obtained after that specified time have either a zero-level of usefulness or have a high rate of depreciation as time moves further from that event before the system gives a response. Such parameter sometimes called *real-time system deadline*.

The same definition using another words.

> *A **hard** real-time system must meet its deadlines with a near-zero degree of flexibility while a **soft** real-time system must meet its deadlines but with a degree of flexibility.*

Operation Systems

O. Tykhomyrov

What is a Real-Time Operating System

The features of a Real-Time System

Design philosophies

RTOSes available on the market

Useful URLs

Hard Real-Time Operating System

The same definition using another words.

*A **hard** real-time system must meet its deadlines with a near-zero degree of flexibility while a **soft** real-time system must meet its deadlines but with a degree of flexibility.*

Considering our first example, it does not mean that all data-base systems are soft real-time systems: a data-base may be a part of a hard real-time system, and its response may become a crucial part of the overall reaction time of the system. An assembly line also may be considered as a hard real-time system if the calculation could not be performed for making the object available at the specified time, the operating system would terminate with a failure.

Two Kinds of Real-Time OSes (1)

• When we deal with data-base systems, we usually mean interactive way of working. The corresponding RTOS is called **interactive** RTOS. This definition *used by commercials*, it means that a system calculates needed data within invisible for a human period of time.

.

Two Kinds of Real-Time OSes (1)

- When we deal with data-base systems, we usually mean interactive way of working. The corresponding RTOS is called **interactive** RTOS. This definition *used by commercials*, it means that a system calculates needed data within invisible for a human period of time.
- In our second example we have the system usually we are interested more. Such systems control the behaviour of some apparatus, machinery, or even an entire factory. We call these real-time **control** systems.

.

Two Kinds of Real-Time OSes (1)

- When we deal with data-base systems, we usually mean interactive way of working. The corresponding RTOS is called **interactive** RTOS. This definition *used by commercials*, it means that a system calculates needed data within invisible for a human period of time.
- In our second example we have the system usually we are interested more. Such systems control the behaviour of some apparatus, machinery, or even an entire factory. We call these real-time **control** systems.
- The border between these two kinds of RTOSes is quite invisible, because a human might rule, interactively, a control system (e.g. a pilot of a modern craft).

.

Two Kinds of Real-Time OSes (1)

• When we deal with data-base systems, we usually mean interactive way of working. The corresponding RTOS is called **interactive** RTOS. This definition *used by commercials*, it means that a system calculates needed data within invisible for a human period of time.

• In our second example we have the system usually we are interested more. Such systems control the behaviour of some apparatus, machinery, or even an entire factory. We call these real-time **control** systems.

• The border between these two kinds of RTOSes is quite invisible, because a human might rule, interactively, a control system (e.g. a pilot of a modern craft).

Usually when we're talking about a RTOS, we mean a real-time control system.

Operation Systems
O. Tykhomyrov
What is a Real-Time Operating System
The features of a Real-Time System
Design philosophies
RTOSes available on the market
Useful URLs

Two Kinds of RTOSes-2

There are the more common way to divide RTOS into two separate classes.

Embedded systems: the controlling microprocessor is an internal part of the entire product, invisible to the end user and where the behaviour of the system is factory defined. The user can only issue a very limited and pre-defined set of instructions, usually with the help of switches, push-buttons and dials. (Example: a washing machine) The microprocessor included in the system has been programmed in the factory and cannot be reprogrammed by the user. It's also very important, that embedded systems are computing systems with tightly coupled hardware and software integration, that are designed to perform a dedicated function.

Two Kinds of RTOSes-2

There are the more common way to divide RTOS into two separate classes.

The systems that make use of a normal computer, where a human being can follow in some detail how the controlled process is behaving etc. The essential difference with an embedded system is that a system in this second class can be entirely reprogrammed, if desired. Also, the computer is not necessarily dedicated to the controlled process, and its *spare capacity* may be used for other purposes.

Two Kinds of RTOSes-2

So, we also have embedded and normal Real-Time Operating Systems

Characteristics of Real-Time System

- Now we can easily guess the main characteristics of a Real-Time System.

Firstly, the RTOS must produce correct **computational** results, called **logical** or **functional correctness**, and

secondly, these computations must conclude within a *predefined period*, called **timing correctness**.

- It means the overall correctness of the system depends on both, functional and timing correctnesses, and the second one is at least as important as the first one.

Yet Another Definition

Sometimes you can meet some very special definition of 'real-time' for special proposal. For instance, real-time for digital signal analysing. If generating and/or analysis can be done within the same interval of time as analysis+generation the same data without digital analysing, this is real-time. For instance, if 2.01 seconds is needed for analysing sound signal of duration 2.00 seconds, it is not real-time but if the same analysing can be done within 1.99 seconds, this is real-time.

Definition Summary

So, it is impossible to find very classical definition of real-time systems that should be known as a child a Nursery rhyme. There are many definitions for them in modern computer literature, they are similar to each other.

Definition Summary

The main document you should consult in many cases, is the POSIX 1003.1 standard (POSIX is an acronym for **P**ortable **O**perating **S**ystem **I**nterface), also know as **The Single UNIX Specification**, 2008 edition. (`http://www.unix.org/2008edition/`).

Definition Summary

POSIX defines real-time for operating systems as the following:

> *Realtime in operating systems: the ability of the operating system to provide a required level of service in a bounded response time*

The classical example, a robot that has to pick up something from a conveyor belt, and many other useful things you can find at
`http://www.faqs.org/faqs/realtime-computing/faq/`

Signalling in a Railway System

Let us take signalling in a railway system as an example. This simple example will give us an idea of what a real-time system is composed of.

Signalling in a Railway System

A railway track is divided into sections. Collisions are avoided by applying a simple principle: never allow more than one train to be present in a section at any moment in time. If a train may or may not enter a section is indicated by a **semaphore** at the entrance of a section.

We consider a system which controls the railway signals for a large territory, so that there are many tracks and many trains, i.e. we have a large-scale system, but it seems simple enough. If there is a train moving forward in section $i - 1$ and section i is free, then the system shall put the signal at the entrance of section i to green and then back to red again as soon as the train has entered section i.

Signalling in a Railway System

So, apart from catering for the signals, the system must keep a "data base", reflecting the situation at any moment: which sections are free and which are occupied.
How do we know that a train is actually entering a section? We could put a contact which closes when the train is on top of it and 'scan' all contacts at regular intervals. A TGV of 200 meter length and running at close to 300 km/h would be on top of the contact for 2.4 seconds. So we must scan all the contacts every 2.4 seconds. *Not very nice*. It is a heavy load, and we will find the vast majority of the contacts open in any case.

Signalling in a Railway System

We could scan only those contacts where we expect a train, to reduce the load, but still we must complete each scan within 2.4 seconds and repeat it at the same rate. We would still have a good chance of missing a lonely locomotive, running at 100 km/h, whereas a slow freight train would "sit" on the contact for several scan periods. And what if a train stops for one reason or another astride two sections?

Signalling in a Railway System

So we better use **interrupts**, generated by a contact, or a photodiode, when *the last wheel has entered* the section. We then *don't lose time* looking at open contacts.
We conclude that we must be able *to sense the inputs to the system*, and, instead of *polling* the inputs constantly, it is often advisable to use *interrupts* to indicate that the state of the controlled system has changed.
The **instrumentation** with **sensors** and **actuators** is an extremely important ingredient of the control system.
We also must have the possibility of *"creating"* a train and *"destroying"* another. A train waiting at the terminus does not *"exist"* yet. Someone or something must tell us that it effectively started moving. The timetable would help, but trains do not always start on time... Also, there may be extra trains, which are not in the timetable.

Signalling in a Railway System

Checking the actual situation against the time table would allow us to predict the arrival time of a train and to send a message to the station master, informing him of possible delays. This may also need some calculation to be done.

A few more ingredients of a real-time system are therefore: *input of operator instructions and output of messages or log data, some calculation on the data and access to a data base*.

We mentioned time. In our case it would be useful to have a **real-time clock**, and for most real-time systems it is **essential**.

Supervisors and managers will certainly want reports, so **data logging and reporting** is yet another task.

Signalling in a Railway System

Then there are potential **conflicts**, which must be solved. So far we assumed that you entered section i, coming from $i-1$. But on a single track railway, you can also enter i coming from $i+1$. There may be a problem determining who arrived first if two candidates want to access section i. The same problem exists when two tracks converge into a single one. The real-time system needs therefore some means of **synchronisation**.

The system is not a closed world: there are railway lines which cross the border. We must then allow for a means of **communication** between systems, to handle the trains that cross the border.

So we have added two more ingredients to our system: **synchronisation** and **communication**.

Signalling in a Railway System

At this stage you feel confident and you start writing your program, in a nice loop, where you execute all these tasks. With the interrupts you have overcome the 2.4 seconds constraint, but still you must **react fast enough** (you don't want to stop *both* of the two trains which rush toward the point where the tracks converge into a single one, do you?). After some trouble you get it all working, and then you realise that there may be **emergencies**. For instance, it happens sometimes that a train loses a few carriages. You better check that a train is still complete when it enters a section. Easy: count the number of wheels, by generating an interrupt for each wheel pair.

Signalling in a Railway System

But: *when do you stop counting?* You'll have to work out some algorithm, using the real-time clock. There is worse: Our TGV would generate 40 interrupts per second, and as there may be 10 or so TGVs entering a new section simultaneously, you may get more interrupts than you can possibly handle. Maybe it is better to count the wheels locally, receive only a single interrupt and then input a number, contained in a register local to the semaphore.
In any case, you must go back to your program and try to fit this in.

Signalling in a Railway System

If you had produced "*spaghetti code*", you will quickly end up screaming. If you had **structured** your program nicely, you will by now realise that your view of the system has drastically changed. Instead of a single, monolithic control program, you distinguish a set of *objects* (trains), using *resources* and *competing for their use*.

Summary (1)

So, using this very simple example we discovered a main set of the real-time operating system features:

1. The system has to be multi-tasking and pre-emptible – task priority has to exist;
2. behaviour of OS should be known – a system of priority inheritance has to exist;
3. the Interrupt Latency (i.e. time from interrupt arrival to start of execution of ISR – **I**nterrupt **S**ervice **R**outine) should be clearly specified, has be compatible with application requirements and has to be predictable;
4. the necessary signalling functions between interrupt routines and taskcodes are handled by RTOS.

Summary (2)

A main set of the real-time operating system features (continue):

5. the RTOS can suspend one task code subroutine in the middle order to run another;
6. the time lag is very small compared to other systems;
7. there are no random time variables, this is good for a direct relationship between instruction and process;
8. tasks are simpler to write;
9. under most RTOSes tasks are simply subroutines.

Process

We have seen that a real-time system has a number of tasks to accomplish. They do not all have the same **priority**. Printing the daily report has much lower priority than handling the interrupts, toggling the signals and avoiding conflicts.

The amount of information to be shared between tasks or to be passed from one task to another, is in general rather limited.

These are two good reasons for handling the various tasks as separate entities: **processes**, which run –more or less– *independently of each other*.

A process is a running (or runnable) program, together with its data, stack, files, etc... The program is just the code, which you may list in source form.

Two Sorts of Design Philosophies

- **Event-driven** (priority scheduling) designs switch tasks only when an event of higher priority needs service, called pre-emptive priority.
- **Time-sharing** designs switch tasks on a clock interrupt, and on events, called *round-robin*.

Two Sorts of Design Philosophies

- **Event-driven** (priority scheduling) designs switch tasks only when an event of higher priority needs service, called pre-emptive priority.
- **Time-sharing** designs switch tasks on a clock interrupt, and on events, called *round-robin*.

Time-sharing designs switch tasks more often than is strictly needed, but give smoother, more deterministic multitasking, giving the *illusion* that a process or user has sole use of a machine.

Two Sorts of Design Philosophies

- **Event-driven** (priority scheduling) designs switch tasks only when an event of higher priority needs service, called pre-emptive priority.
- **Time-sharing** designs switch tasks on a clock interrupt, and on events, called *round-robin*.

Early CPU designs needed many cycles to switch tasks, during which the CPU could do nothing useful, so early OSes tried to minimise wasting CPU time by maximally avoiding unnecessary task-switches. It means Event-driven design.

Scheduling

- There are three states for a task: *running*, *ready*, *blocked*.
- Designing the *right* scheduler is the main road for switching tasks state. Normally, a task is represented by its data structure in the memory of the **ready** list.
- Switching between tasks should be done to minimise the time in critical sections of the kernel, while changing the task status is impossible.
- There are two main approaches to scheduling. The first one is cooperative scheduling while the second is preemptive scheduling. You also can find some extra algorithms, such as *Earliest Deadline First*.

Round-Robin Scheduling

This is a classical algorithm (with many changes and modifications). It assigns time *slots*, or *slices*, to each process in equal portions and in circular order, handling all processes without priority. This is certainly real cooperative scheduling. This is not good for real-time processes but may be used in some combination with other algorithms.

Preemption

Preemptive multitasking use mechanism of an interrupt to suspend the currently running task, or process and invokes a **scheduler** to determine the next process for running and in turn to use machine resources like CPU, memory etc.
In preemptive multitasking, the OS can also initiate a context switch in order to satisfy the scheduling policy's priority, thus preempting the active task. In general, preemption means 'prior seizure of', 'to the capture'. When the high priority task at that instance seizes the currently running task, it is known as preemptive scheduling.

Preemption

If a **kernel** is a monolithic or hybrid, all its processes run in kernel space. It usually means that to initiate a context switch, scheduler has to wait until a driver or other part of the kernel completes its execution and return control to the scheduler. If the kernel allows such kind of switching, we say about *kernel preemption*.

Preemption

There are two main benefits to this method in monolithic and hybrid kernels (in contrast to **microkernel**):

1. A device driver can enter an infinite loop or other unrecoverable state, crashing the whole system;
2. Some drivers and system calls on monolithic kernels are slow to execute, and can't return control of the processor to the scheduler or other program until they complete execution.

Intertask communication and resource sharing

In order to avoid such conflicting situations, the two processes which compete for the common resource (shared memory, for instance), must **synchronise**.

A **test-and-set** instruction could be used to solve the possible conflict of accessing simultaneously a critical region.

But what do we do, if we find that the test failed, because the bit had already been set before by someone else?

We go to **sleep**. Does this solve the problem?

No, it does not. Our process could have been suspended immediately after the failed *test-and-set* instruction, and just before we have had a chance to go to sleep.

The general solution is **Dijkstra's semaphore mechanism**.

A **mutex** or binary **semaphore** is enough to protect **the critical regions of the program**.

Intertask communication and resource sharing

The mutex avoids that one process writes to the common resource while the other is reading it, or that two processes write *simultaneously* to one printer.

Access to the semaphore itself must be **atomic**, that is not interruptable by another process. Otherwise *process 1* may inspect the semaphore and find it green. It then gets swapped out of memory (bad luck...) and *process 2* which has just started running, also inspects the semaphore and naturally also finds it green, puts it to red and starts using the resource. Some time later, *process 1* starts running again and, thinking that the semaphore is green (it had tested it before, remember?!), also starts using the resource...

Intertask communication and resource sharing

Access to a semaphore can be made *atomic* by disabling interrupts, but it is too dangerous to let a user program do this.

On *NIX a *semaphore* is accessed via a system call.

You do not therefore do the testing and signalling of the semaphore separately from each other. You just *signal the semaphore*.

As the semaphore module remembers that the process is waiting on the semaphore, it makes the process runnable again as soon as the resource becomes free and the semaphore is put to green by the other process.

Operation Systems

O. Tykhomyrov

What is a Real-Time Operating System

The features of a Real-Time System

Design philosophies

RTOSes available on the market

Useful URLs

Intertask communication and resource sharing

We also need mechanism of passing messages from one process to another. We can also try to share as much as possible avoiding sending messages, (re)open files etc. In this case we may use *threads*.

Summary

- Special **scheduler**;
- messages and semaphores to intertask communication and resource sharing;
- memory allocation;
- temporarily masking/disabling interrupts;
- time controlling *(should not forget!)*;
- probably more. . .

There are so many of them

In fact, there are *plenty* of different RTOSes. Using WIKIPEDIA, you can find a list with something like **60** OS names that pretends to be real-time ones. **GNU** world suggests less numbers, and many of them based on LINUX or BSD.

The preliminary list

Here you are a very short selecting from the huge list. Consult to WIKIPEDIA.

Name	Base	Type	State	Platform
Contiki	BSD	embedded	active	MSP430, AVR
ChibiOS/RT	GNU GPL	embedded	active	ARM7,AVR
RTLinux	GNU GPL	general	active	As LINUX
Ubuntu Studio	GNU GPL	multimedia	active	As LINUX
MontaVista	Monta Vista	embedded	active	As LINUX
LinxOS	Linux Works	embedded	active	Motorola 68010
QNX	RT Systems since 12.2007	general	active	MIPS,PowerPC StrongARM

x86-platforms are supported almost by any kind of *general propose* RTOS.

Operation Systems
O. Tykhomyrov
What is a Real-Time Operating System
The features of a Real-Time System
Design philosophies
RTOSes available on the market
Useful URLs

What to Do

- Use **Open Source Software**. It means you may **modify** the kernel code.
- Try to *measure* the system parameters. Never trust information from the vendor you have selected! The keys for searching the information is *Rate Monotonic Scheduling*
- Try to test the kernel first with a CPU you are going to use.
- In case of using non-GNU programs, try to read licence carefully!

Weak points of LINUX

For real-time applications you may want to perform **asynchronous I/O**. This means you want not to block on an I/O operation, but **continue processing** immediately after making the I/O system call. The user process should then itself determine when the I/O operation is finished. The Linux kernel has provision for this, but standard device drivers do not use it.

Another point to note is that to **communicate with exotic devices**, you need to write a device driver. For UNIX this means re-building the kernel. Linux allows to load a module, such as a device driver and link it to the kernel at run time.

Weak points of LINUX

- pre-allocation of large files.
- Fine-grained real-time clocks.

Part of the **pthreads** package, if the hardware is capable, can help to solve this problem.

Useful URLs

- http://en.wikipedia.org/wiki/Real-time_operating_system
- http://people.mech.kuleuven.be/~bruyninc/rthowto/
- http://www.lynuxworks.com/
- http://www.onesmartclick.com/rtos/rtos.html