You need an RTOS!

What is the Challenge?

Nowadays, embedded systems and more generally cyber-physical systems (CPS) are present everywhere in our daily lives. Computers embedded in planes and trains, wearable computing objects and medical devices are only a few examples of such technology-intensive devices in our modern society. As many human lives or huge financial investments often rely on these critical systems, they cannot experience failure of any kind.

Critical systems must operate in situations where any failure can have from annoying to dramatic consequences. Unlike a PC, a cruise control system does not have a reset button! The system must be **"very" reliable**, e.g., the probability of failure must be extremely improbable. Most CPS are thus **"real-time".** This is not just being "fast", but being **"always on time"** in a very precise way. Being late is as bad as being wrong. In a braking system or in an autonomous drone, we cannot afford any delay in reacting to an event without hitting a wall or falling on the ground. Finally, embedded systems often have limited resources (e.g. memory) and must operate under constraints that need to be optimized (e.g. use low power). Meeting the deadlines can be crucial for the system (i.e., the deadlines are considered to be *hard*) in order to avoid catastrophic consequences. Examples include anti-lock braking system (ABS) avoiding uncontrolled skidding or aircraft control. But very often in the applications, while deadline misses can be tolerated (i.e. the deadlines are soft or *firm*), their amount and frequency must be limited for system correctness or for system Quality of Service (QoS). Examples include video streaming (bound the packet loss ratio) or digital audio processing (conversion, compression and routing) in order to bound the latency for instance.

So, what are the requirements for a modern embedded system?

The system must be **highly efficient** and provide high computing performance under strict constraints such as reduced **size and power consumption**. It must be **highly reliable**: failures must be "extremely improbable" and the system must be fault-tolerant. The system must respect **real-time constraints** and execute all computations **predictably** in a **timely manner** without any delay. Finally, it must be **secure** and enforce protection against illegitimate use and access.

Why an Operating System? Why not Bare Metal?

As computer systems become more and more complex, their control and application software grows larger and more complex. It is not possible for every application programmer to know and understand all these technical details and specifications: the **effort would be too large** to produce the code and the **resulting quality would be too low**. Moreover, the various and often limited hardware **resources must be managed efficiently** (often optimally) and this is a very challenging problem. This is why it is very difficult and costly to build applications from scratch for any target hardware but the simplest microcontrollers. The bare metal "single control loop" system lacks the ability to introduce fail-safe, recovery or smooth degradation features, to have flexible modes of control, etc. Basically, it introduces "Single Point of Failure". The **"bare metal" paradigm is no longer an acceptable, feasible and cost-effective solution**. In the following we will present rational arguments in favor of Operating Systems.

The **Operating System** (OS) is a software that acts as intermediary between applications and computer hardware.

The OS provides a development environment for applications that aims to be an efficient **resource manager** as well as an **elegant hardware abstraction**.

The concept of operating system is not new. Early OS were introduced for general-purpose computers in the 1960's. Later they evolved into systems like DOS, MS-Windows, and modern systems such as Android, Linux or Mac OS X are familiar.

As an abstraction layer the OS hides the hardware complexity. The abstraction includes the notion of processes (abstraction of the CPU), files (abstraction of the disk) and virtual memory (abstraction of the RAM), etc. The OS is the **"platform".** Programs written for a given OS will run on any hardware architecture supported by that OS. Bare metal applications on the other hand have to be significantly rewritten to work properly on new hardware. Porting legacy code on a new processor is therefore costly and time-consuming. Using an OS makes this effortless and makes it easy to leverage new hardware features brought by the latest generations of embedded devices.

As resource manager the OS includes advanced data structures and algorithms for resource sharing, process scheduling, memory management, context switching, I/O managements and interrupt handling.

These features provide a higher-level abstraction to the application programmer by means of efficient and elegant concepts, drastically reducing the apparent complexity of the hardware and therefore enabling more complex and more reliable applications.

Why not a General Purpose OS? Why an RTOS?

In embedded systems, GPOS are usually poor choices.

GPOS are not very reliable: the problem with operating systems such a MS Windows and Linux is that they do not satisfy the above requirements. High performance is not the issue: there are high performance versions of those OS. But **reliability** is by far not enough for any mission-critical or safety-critical embedded system. To be used in those embedded systems, the OS must be "very" reliable, e.g., the average probability per flight hour for catastrophic failure conditions must be less than 10^{-9} or "extremely improbable". Testing is not enough; quality must be proven. GPOS contain millions of lines of code and it is virtually impossible to prove that they are reliable enough. **Reliability should be built-in "by design".**

GPOS	RTOS
optimize the **average case**	optimize the **worst case**
maximize **throughput**	minimize **latency**
schedule fairly	schedule timely
best effort	real-time & **predictable**
dynamic programs open environment	specific programs closed environment
Unsecured	secure

GPOS are neither real-time nor predictable: GPOS are "best effort" in that they execute programs **fairly**. They are designed to optimize the average execution time of jobs and not to bound the worst case execution time. GPOS do not give any guarantee concerning the limitation of delays or meeting deadlines.They also use many "dynamic" (i.e. non deterministic) programming features. GPOS are not designed for real-time predictable behavior.

GPOS are not fully secured: some GPOS are more or less "secured" but their size, variability and "openness" makes them **vulnerable to security threats**. Having the security required for "trusted computing platforms", such as those in money cards, also must be built-in by design.

GPOS are "too large" and "use too many resources" for many embedded platforms. The reason is mainly that they need to have features that an embedded system does not need, such as elaborate Graphical User Interfaces, they need to run a large variety of user programs and support many hardware devices. An embedded system only needs to support a few specific programs required for a given application, and must only support a few devices on the target hardware. Being too large also means it's virtually impossible to prove that they are correct.

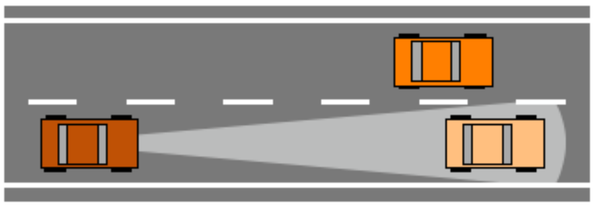
For all the above reasons, since the late 1970's a new kind of OS has been created: the Real-Time Operating Systems (RTOS).

RTOS are reliable and predictable: they are designed to **bound** the maximal duration of **critical**

operations. Critical operations include OS calls and interrupt handling. The designer optimizes the system behavior for the **worst case.** Some RTOS are even **"certifiable"** to satisfy the strictest safety norms, e.g. ISO-26262 provides a standard for functional safety management for automotive applications or DO-178B a software certification standard for airborne systems on commercial aircraft. The RTOS manages the resources and schedules the programs timely and minimizes the latency. Specific and **advanced process schedulers** (e.g. Rate Monotonic or Earliest Deadline First) are required to **"prove"** that the system is **always on time** while general purpose schedulers (typically Round Robin of Unix-like and MS Windows systems) cause deadline misses even for weakly loaded systems.

Why can't we use General Purpose Schedulers? Why Real-Time Schedulers?

Example: Adaptive Cruise Control



Consider a road vehicle cruise control that adjusts the vehicle speed to maintain a *safe* distance from the front vehicle (the brown car "follows" the peach one on our schema). An on-board sensor (radar or laser) *periodically* determines the distance and the speed of the front vehicle; based upon those variables the system adjusts automatically the speed of the following vehicle. Of course the driver can, at any time, brake and disable this automatic system.

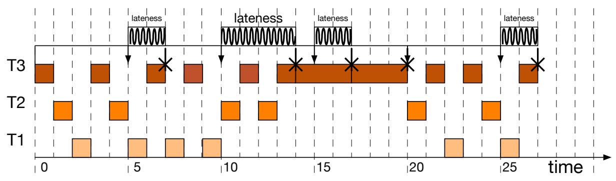
The Model

In this illustrative example we will consider, for the sake of simplicity, a system composed of three real-time periodic tasks: T1 displays the front vehicle distance, T2 adjusts the car speed and T3 disables the system (user brake). T3 must be very reactive and repeats every 5 milliseconds, T2 is also quite critical and repeats every 10 milliseconds while T1 is not crucial for the system safety and repeats every 20 milliseconds. Each task corresponds to a difference piece of code executed periodically. In the following, each task instance is called job. Job releases and deadlines are represented using arrows on the following figures. Let us assume, for the sake of simplicity, that for each task the deadline corresponds to the next release. Period, deadline and execution time (actually an upper-bound on the execution time) are summarized in this table.

Task	Execution time (ms)	Period & deadline (ms)	Load
T1-displaying	4	20	20%
T2-adjusting speed	2	10	20%
T3-disabling	3	5	60%
			100%

The goal of General Purpose Schedulers is to *fairly* share the processing resource — e.g. the CPU – and provide good average process response time. The key idea is to assign time slices to each process in equal portions and in a circular order. This is Round Robin scheduler, a typical scheme for most GPOS.

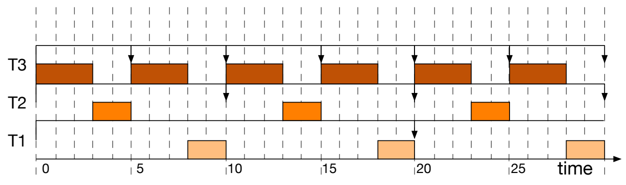
Round Robin



Round Robin does not take into account process timing requirements nor deadlines. Given our three tasks, Round Robin will assign 1 quantum every three quanta (at least while the other two tasks are active). For the sake of simplicity, we will assume that the quantum duration is 1 millisecond. That schedule is catastrophic for critical tasks like T3. In the best case Round Robin assigns the red quanta to T3. In that case T3 misses a large portion of its deadlines and the lateness for the five first jobs are 2, 4, 2, 0, 2 milliseconds, respectively.

Meanwhile there exists a simple real-time scheduler, RM, that feasibly schedules the same system.

Rate Monotonic – RM



The Rate Monotonic (RM) is a preemptive scheduling algorithm used in real-time operating systems (RTOS) with a static-priority scheduling class. The static priorities are assigned on the basis of the cycle duration of the job: the shorter the cycle duration is, the higher is the job's priority. In our system, and according to RM, T3 receives the highest priority, T1 the lowest priority and T2 middle priority. This time all tasks are on time using RM (notice that the schedule repeats every 20 milliseconds) while Round Robin requires a faster CPU in order to meet the deadlines. The RM real-time scheduler has an outstanding property: RM feasibly schedules any system not overloaded, i.e., workloads not larger than 100% (at least for harmonic task periods like in our system where all task periods are multiple of 5 milliseconds). In real systems this outstanding property must be adapted carefully since the system is preemptive and we cannot neglect the switching time and system overhead.

What's next

If you want to know more about Real-Time Operating Systems, the next paper in this series is "Challenges for Next Generation RTOS" White Paper #2. The paper addresses advanced concepts and challenges for Next Generation RTOS: system overhead (e.g., preemption delay), modern multicore hardware platforms, multi-threading and parallelism issues, advanced real-time scheduling algorithms.

References

1. "Real-Time Systems", J.W.S. Liu, Prentice Hall, 2000.
2. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", C.L. Liu and J.W. Layland, Journal of the ACM, Volume 20 Issue 1, Jan. 1973.
3. "Multiprocessor scheduling for real-time systems", Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. Springer, 2014.

Joël Goossens, PhD
HIPPEROS Chief Scientific Evangelist

Joël Goossens is Professor at the Université libre de Bruxelles, Belgium (ULB), since October 2006. He founded and chairs the "Parallel Architectures for Real-Time Systems" research group. His main research interests are presently in real-time scheduling theory, real-time operating systems and embedded systems.

Joël Goossens is co-Founder of the HIPPEROS company and serves as Chief Scientist. His main activities concern the R&D, research projects and diffusion of innovations.

HIPPEROS S.A.
Predictable Real-Time, Proven Performance
Rue A. Piccard, 48, B-6041 Gosselies, BELGIUM
www.hipperos.com

esa business incubation centre Redu