Literature review

Introduction

Applications of RTOS are multiple, especially in areas where reliability is a key aspect, even with radically different purposes. These systems are typically found in astronautics, mainframes, aeronautics, robotics, or embedded systems in general (and IoT in particular) [**Buttazzo, 2011**]. They are typically more tolerant to failures and more predictable [**Aroca & Caurin, 2009**], and particular care is taken about the implementation and the tests.

However, new possibilities recently arose with the emergence of new tools centered around solving well-defined problems with commonly-used techniques. This brings us to Rust: originally created by Graydon Hoare, who worked at the Mozilla Foundation, it is now developed by the Rust project developers, centered around the language's Github repository. Designed to be fast and reliable, this language is memory safe, uses Resource Acquisition Is Initialization (RAII) to manage memory (avoiding the necessity of a garbage collector), supports polymorphism and natively supports concurrency [**Matsakis & Klock, 2014**].

Whereas the majority of current RTOS have been conceived when Rust did not exist, they are written in languages that show their limits when it comes to real-time computing, or are simply stuck in the past [**Burns & Wellings, 2001**]. Hence a RTOS implemented in Rust seems very promising. Very few attempts in this area have been made [**Heldring, 2018**], and yet RTC is a growing field in the need for a new generation of RTOS aimed to solve arising challenges.

RTOS case studies

VxWorks

Initialized in 1987, VxWorks is a proprietary RTOS developed by Wind River, whose kernel is monolithic. The OS is delivered with an IDE, based on Eclipse. It is primarily used in embedded systems, with a recent progress towards IoT. Uses examples include NASA Mars rovers, Boeing AH-64 Apache attack helicopter, and Olympus Corporation's surgical generator. VxWorks's scheduler is preemptive and based upon the round-robin scheduling policy (therefore it is a time-sharing RTOS). It is written in C++ and supports Java applications. VxWorks has been proven to be very deterministic [**Ip, 2001**], but less performant than other alternatives [**Barbalace et al, 2008**].

Spring

The first stable version of Spring has been released in 1993. It is written in C++ and inspired by Mach, a research microkernel developed at Carnegie Mellon University and meant to run on a network of processors (related topic: distributed computing). The member of the projects clearly wanted to solve the problems related to the design of RTOS [**Stankovic & Ramamritham, 1989**]. The scheduling algorithm guarantees jobs to be run if they have been accepted, and the environment allows on-the-fly modifications of the constraints and reacts dynamically according to the changes [**Stankovic et al, 1999**]. However, Spring is not multithreaded; it is small enough to run properly on a single thread, but it might constitute a serious drawback for up-scalability.

RTLinux

Written in C and originally developed by Victor Yodaiken, Michael Barabanov, Cort Dougan and others, RTLinux became a product of FSMLabs and then Wind River (cf. VxWorks). Basically, it supports a guest Linux OS as a preemptive task using paravirtualization. Designed to be modular, its simple priority scheduler can be modified to fit a specific purpose [**Barabanov, 1997**]. The jobs have direct access to the hardware and a high level of autonomy: when launched, deadline, period, and release-time constraints are transmitted to the RTOS, but yet an informed analysis of the scheduling feasibility before accepting jobs seems difficult [see: Spring].

QNX

QNX is a proprietary Unix-like RTOS developed by Blackberry Ltd. and aiming to run on embedded systems. First released in 1986, it is still active nowadays. Minimal (scheduler, IPC, interrupts and timers), QNX has been ported to a large number of platforms. Its IPC (including I/O) is based on task priority, which is uncommon enough to be noted [**Hildebrand, 1992**]. The scheduling policy is priority-preemptive and supports adaptive partition scheduling, to guarantee a minimal amount of resources to a group of tasks [**Dodge et al, 2005**]. It can run on multiprocessors systems; similarly to VxWorks, it shows a high reliability and determinism but is less performant [**Aroca & Caurin, 2009**].

RTAI

The product of the fusion of the RTOS Xenomai and the early RTAI project, RTAI is born is 1999, apparently from a scission regarding the problems and future evolutions of several other projects [**Mantegazza, 1999**]. It is a community-based extension of the Linux kernel that allows deadline constraints for tasks and a deterministic response to interrupts. The latest stable version has been released in 2018, proving the project's vitality. Compared to VxWorks, RTAI is slower (at least when many interrupts occur in the environment) but has a better message latency, under certain conditions of implementation [**Hambarde et al, 2014**]. The main advantage of RTAI is its integration in the Linux community, a potential source of free already-written applications [**Aroca & Caurin, 2009**].

IPC

As a core feature of an OS, IPC is a very important topic. It is possible to divide IPC between synchronous (blocking) and asynchronous (non-blocking) communication [**Sundell & Tsigas, 2002**], depending on the needs. Since asynchronous application allows calling program to pursue its execution, a proper use of this technique could prevent task starvation, and is therefore highly desirable. RTOS serving in distributed systems have a strong need for reliable IPC, for they have to avoid network [**Peterson et al, 1989; Rajkumar et al, 1995**] and synchronization [**Abraham et al, 1990**] issues in addition to more classical concerns. Note that this last point is out of the scope of this project, for aimed to run on a single machine.

Microkernel

Formally described long ago [**Hansen, 1970**], microkernels include at least a virtual memory manager, a scheduler and an IPC mechanism. Further characterization of the essential components has also been shown [**Kirsch et all, 2005**]. Although this design is widely accepted, it is only a theoretical definition, that does not prevent microkernel OS to encompass more features, or to provide interfaces for extensibility [**Bershad et all, 1994**], bringing a certain scalability to the system. With the same idea, running non- real-time applications on a guest OS, by virtualizing the hardware through interfaces, is possible [**Nelson et al,**

2014]. The choice of a microkernel is often related to embedded devices, which by nature are more limited in resources, in terms of computing [**Zuberi et all, 1999**] or electrical power [**Osman & Koren, 2003**].

Scheduling

There are two main approaches when it comes to scheduling: preemptive scheduling and earliest deadline first. We will explore them in the next sections. Note that other approaches such as stochastic digraphs with multi-threaded graph traversal and cooperative scheduling can also be found.

Preemptive scheduling

The tasks can be interrupted by another task with a higher priority or the scheduler itself, exception of the kernel tasks running in kernel mode which are usually not preemptible. So as not to be a bottleneck in systems where tasks are often preempted, context switches must be as fast as possible. The concept is itself divided into several derived scheduling policies, among which the most widespread are:

- rate-monotonic scheduling (RM): proven to be optimal in an environment of periodic tasks with static execution times [**Liu & Layland, 1973**], it assigns static priorities to tasks
- round-robin scheduling (RR): each task is allocated a fixed quantum of time, during which it is guaranteed to have access to a resource over all the other tasks [**Kleinrock, 1970**]

Preemptive scheduling can sometimes produce priority inversion hazards, which can be avoided by implementing priority inheritance [**Sha et al, 1990**], even it seems to raise deeper problems [**Yodaiken, 2004**].

Earliest deadline first (EDF)

Dynamic priority algorithm based on a priority queue (a container that can be implemented in nearly all imperative languages). On a regular basis, and/or the occurrence of a particular event, the algorithm computes the priorities of the tasks in connection with their deadlines. It has been shown to reach a global optimum and capable of achieving full processor utilization [**Liu & Layland, 1973**].

Case studies

According to a benchmarking study [**Buttazzo, 2005**], the processor utilization of EDF makes it more desirable for embedded applications and more responsive to aperiodic tasks, which tend to degrade predictability. The advantages of RM seems to mostly apply to high priority tasks. Modifications have been made to solve those problems and improve the two algorithms [**Buttazzo & Stankovic, 1995; Koren & Shasha, 1995**].

Virtual Memory Management

The virtual memory is an abstraction provided by the OS to the programs and is a key feature for ensuring process isolation and other safety concerns. The virtual memory manager is responsible for load the needed data so as to be accessible to the processes as fast as possible. Despite the growth of the RAM and cache components in terms of available space, memory consumption increased as well, and the need to adequately manage data in fast memory units remained. It is worth mentioning that computers did not include this functionality until the early 1970s; we can cite THE multiprogramming system as a notable exception [**Dijkstra, 1967**], along with a few others. There are two ways to organize memory, paged

memory and segmented memory, each resulting of a different point of view, that can also be combined [**Denning, 1970**].

Paged memory

An organization based on blocks of contiguous addresses (page sizes can be variable, even inside an OS, but are typically 4kB nowadays). The addresses are translated using page tables and the pages themselves are managed by a paging supervisor including a page replacement algorithm. It is possible to pin memory pages, meaning they cannot be backed to a secondary storage, in order to guarantee a fast memory access to the OS processes and/or provide a fast IPC mechanism [**Tezuka & al, 1998**].

Segmented memory

The memory is divided into segments that fit the processes logic. Addresses consist of a segment number and an offset within the segment. It is possible for programs to share segments, creating an IPC mechanism as fast as pinned memory (see above) that can be used by regular tasks [**Englander, 2003**]. It is possible to provide a fine control of the memory by associating permissions to segments.

Segmented paging

In this solution, each segment is bound to a page table address. Increasing the memory available can be simply done by adding another page to the segment's page table. It is often cited as the best approach. [**Denning, 1970**]

Page replacement algorithms

In a paged memory system, the page replacement algorithm is responsible for swapping memory pages to be allocated when a page fault occurs. Although there have been attempts to provide generic interfaces for page replacement algorithms [**Rashid et al, 1987; Abrossimov & Rozier, 1989**], the theoretically optimal page replacement algorithm has been found [**Belady, 1966**]. The other existing algorithms are Not recently used (NRU), First-in, first-out (FIFO), Second chance, Clock, Least recently used (LRU), Random, Not frequently used (NFU), Aging and Longest distance first (LDF). Those algorithms will not be studied in detail here. LRU-k, a derivate of LRU [**O'Neil et al, 1993**] may be a good choice for a RTOS, by its good use of Bayesian formula [**O'Neil et al, 1999**]. Derivates from Clock can also show good results [**Carr & Hennecy, 1981**]. Surprisingly, the Random algorithm is not a mediocre candidate and might be considered as a reasonable choice for a first virtual memory manager implementation. Finally, comparative of the most used page replacement algorithms make clear that beyond the differences in terms of performance or memory consumption in particular cases, the choice of an algorithm mainly depends on the environment [**Chavan et al, 2011**].

Rust (vs other languages)

Since Rust is quite a new technology and still maturing, very few scientific papers can be found. A part of the literature emanate from the developers of the project, and describe its design and operating; thus Rust has been guaranteed free from common problems related to memory (overflows, pointers) [**Anderson et al, 2016**]. Another part deal with implementations of various systems in Rust. The most important is without a doubt an implementation of an RTOS in Rust [**Heldring, 2018**], which is closely related to this project. However, a more attentive looks reveal that it is possible to go beyond, especially by adopting a Rust-only axis and a full utilization of the language's features. In the conclusion, the author is confident that "it is

possible to build an RTOS in Rust with competitive performance to C or C++ RTOSes", and this is clear that this project aims to prove.

Summary

We have seen that despite its seniority, the real-time computing ecosystem is growing, carried by new technologic and market trends. However, the current environment is partially slow down and bridled by the use of old tools that accompanied its emergence.

This is why we propose a new implementation of a minimal RTOS in Rust, a promising and yet powerful language. The project will include a functional kernel, an IPC mechanism (primarily based on signals), a dual-level scheduler and a virtual memory management system.

All these features can be extended once the requirements have been met. New components, such as I/O handler, a bootloader, and a rudimentary filesystem could also be added.

This project aims to show that a modern RTOS could not only match the existing ones' performances but bring a real added value concerning the reliability/safety, without additional overhead (especially in terms of speed and response delay), while using long-experienced design specifications.