A comparison between the scheduling algorithms used in RTLinux and in VxWorks - both from a theoretical and a contextual view

Authors and Affiliation

Oskar Hermansson and Stefan Holmer studying the third year of Computer Science and Engineering at the University of Linköping. We can both be reached through e-mail:

Stefan Holmer, steho564@student.liu.se

Oskar Hermansson, oskhe171@student.liu.se

Abstract

The purpose of this report is to inform fellow students of what the differences between the scheduling algorithms in RTLinux and VxWorks are, and when it's appropriate to use them. We describe the two scheduling algorithms used in RTLinux, namely Earliest-deadline-first (EDF) and Rate-monotonic (RM). In VxWorks we describe round-robin (RR) and preemptive priority-based scheduling. We also discuss the POSIX scheduler, which is available both in RTLinux and VxWorks.

A comparison is made where we deal with both the theoretical and contextual view. Finally we conclude that RTLinux and VxWorks uses basically the same scheduling idea but with RTLinux having the possibility of using more advanced algorithms, which though leads to problems with frequent context switching, and on the other hand VxWorks being little bit more predictable and still not suffering from too frequent context switching.

Introduction

This this report is a comparison between the scheduling algorithms in two real time operating system. We have compared RTLinux and VxWorks. We begin by identifying the scheduling algorithms used in each real time operating system. We describe them technically and try to comment on what's good and what's bad with each of the algorithms. In later sections we compare the algorithms used in RTLinux and VxWorks both from a strict theoretical view and from a more contextual view. In the last section we make conclusions and summarize the report.

Body

In this section we will first describe the schedulers available in both RTLinux and VxWorks. Later we compare the real time operating systems by comparing the scheduling algorithms available in a theoretical and a contextual view. In the fourth chapter we come up with conclusions on when RTLinux is appropriate to use, and when to choose VxWorks. We also discuss whether one of the real-time operating systems can be said being better than the other.

1 RTLinux

RTLinux is not an operating system by itself, but rather an extension to already available Linux distributions, which adds real-time functionality. The real-time operating system application which RTLinux adds is called RTCore. RTCore takes over interrupt control from the client Linux OS and has its own real-time scheduler. Though, the client Linux OS still has it's own scheduler and schedules it's own processes. This means that the client Linux OS doesn't know about RTCore and can't preempt it. [1]

Because RTLinux is module based, it's possible to load it into the Linux kernel whenever you need it, and unload it when it's not needed anymore.

To prevent Linux from disabling interrupts, RTLinux replaces the assembler calls cli, sti and iret with S_CLI, S_STI and S_IRET and in that way is RTLinux able to choose how interrupt disabling is supposed to be handled without loss of real-time precision. [2]

According to an article written by Ismael Ripoll the default RTLinux scheduler is preemptive, has a fixed priority and handles normal, non-real-time, Linux processes with lowest priority. This means that if real-time processes need all of the CPU, it will seem like the system has hung because the non-real-time processes aren't running.

Victor Yodaiken from New Mexico Institute of Technology writes that the default scheduler which comes with RTLinux lacks protection against impossible schedules which can lead to starvation. But because RTLinux is module based, it's possible to improve the real-time scheduling in RTLinux by loading a different scheduling module, thus using another algorithm. [3] Other scheduling algorithms available for RTLinux are earliest deadline first and rate-monotonic scheduling. [4]

The possibility to easily change scheduling modules makes it easy to test different algorithms to see which works best for the specific environment it is used in.

1.1 Rate-monotonic scheduling

Rate-monotonic scheduling sets static priorities to the periodic tasks which it's scheduling. If a task has a short period - that is, it will be executed often, the scheduler will give it a higher priority. Tasks with longer period gets a lower priority. If the CPU is executing a low priority task when it's time to execute a task with higher priority, the scheduler preempts the running task and starts executing the high priority task. If a set of periodic tasks can be scheduled by assigning static priorities, the rate-monotonic scheduler will be able to do it. Worse is that rate-monotonic scheduling can't always guarantee that it's possible to schedule a set of tasks, or more correctly, it can't always maximize the CPU utilization. If the CPU utilization required to schedule n tasks overrides $2(2^{1/n}-1)$ the scheduler can't guarantee they will all meet their deadlines. [4]

1.2 Earliest-Deadline-First scheduling

A big difference between Earliest-Deadline-First scheduling (EDF) and Rate-monotonic scheduling is that EDF has dynamic priorities. The priorities are dynamic in the way that the task with the earliest deadline always has highest priority and will preempt any other tasks running. The processes scheduled by EDF scheduling don't have to be periodic or require a constant amount of CPU time. Theoretically EDF scheduling is optimal, which means it can schedule processes so that they will meet all of their deadlines and at the same time make use of all the CPU time available [4].

1.3 POSIX.1b scheduler implementations

RTLinux also offers implementations of the POSIX.1b schedulers. These are SCHED_FIFO and SCHED_RR, a first-in first-out scheduler and a Round-robin scheduler [4]. POSIX.1b makes it easier to write programs which are easier to port to different real time operating systems which also has a POSIX.1b implementation. The idea behind first-in first-out scheduling is to offer the CPU to the task which is in front of the FIFO queue until it terminates. There is no time slicing between tasks with equal priority, which is the case with the round-robin scheduler. Round-robin scheduling is further explained in chapter 2.1, but the general idea is to divide the time line into time slices and running each task for the length of a time slice.

1.4 Summary

Initially RTLinux has a quite primitive, preemptive and fixed priority scheduler which does not have protection against impossible schedules (may lead to starvation). Therefor we think that the programmer has to be sure not to put the OS in starvation and not to schedule more processes than the CPU can handle to still manage to complete them in time. And it's much harder for the programmer to know whether the tasks will be executed before deadline or not. If the OS is going to run in a very time critical environment it will probably be desired, if not also needed, to use a more fancy scheduling algorithm.

We think the possibility to use the more complex algorithms rate-monotonic scheduling or EDF scheduling makes RTLinux much more useful. If you need to be totally sure that deadlines are met you can use EDF scheduling or, if your processes are periodic, you can use rate-monotonic scheduling and calculate how low your CPU utilization must be.

2 VxWorks

This chapter will shortly describe VxWorks in general and describe what possible scheduling algorithms are available and how they work.

VxWorks is a commercial real time operating system created in the early 1980's and developed by Wind River. It was used in the two rovers used to explore the planet Mars in 2004. VxWorks uses a micro kernel with a minimal of utilities. Everything but the basic functions are kept outside of the kernel to allow smaller footprint, which is important when using in embedded systems. VxWorks was designed with predictability in mind which affected their choice of scheduling algorithms. [4] [5].

VxWorks's default scheduler comes with two possible scheduling algorithms [http://www.windriver.com/vxworks/technology.html]: A preemptive priority-based scheduler (with 256 priority levels) and a round-robin scheduler. It is also possible to use a POSIX scheduler with both a first-in first-out (FIFO) and a round-robin algorithm. [4].

2.1 Round-robin

Round-robin is a scheduling algorithms that divides the time line into equally big time slices and then every task is run for the length of a time slice. The scheduler cycles through all tasks in the ready queue so that every task gets the same amount of CPU time. If a task is not finished by the end of the time slice, it is preempted by the scheduler and moved to the end of the ready queue. If a task ends before the end of the time slice, it yields the CPU and next task in the ready queue is run (still only for at most the length of a time

slice). This scheduling algorithm is used when all tasks are equally important.
The behavior of round-robin depends on how big time slice is chosen. A big time slice, where most tasks finishes before the time is up, will result in a behavior similar to non-preemptive first-come first-served. A small time slice will result in a lot of context switching. Depending on the CPU design context switching all the time could result in a lot of overhead. [4].

Pro's:

- Starvation-free (since every task does get CPU time eventually every task will finish)
- The size of the time slice is set by the user which can be used to modify the behavior of the algorithm

Con's:

- No way to prioritize important tasks (or tasks with short deadline)

The round-robin scheduling algorithm in VxWorks is not an independent algorithm, instead it is used as an option to the preemptive priority-based algorithm (described below) used to handle tasks with the same priority.

2.2 Preemptive priority-based scheduling

Each task is assigned a priority level and the scheduler always prioritize the task with the highest priority level. The most important task always reside in the CPU. Whenever a task with a higher priority level enters the ready queue the current task is preempted by the scheduler and put back in the ready queue. The new task (with the highest priority) enters the CPU and is run. VxWorks currently support 256 priority levels. Priority level 0 is the highest possible and priority level 255 is the lowest possible.

If there are several tasks with the same priority level in the ready queue the scheduler can use the round-robin scheduling algorithm, described above, to switch between the tasks. If the tasks are equally important it's fair to run them equally much. With several tasks with the same priority it is also possible to use a non-preemptive first-come first-served (FCFS) algorithm meaning that a task is never preempted by another task with the priority level (only by tasks with higher levels). Using this is the default option in VxWorks.

To prevent a task from missing it's deadline the priority level of a task can be increased after time (when the deadline is approaching). With a higher priority level it's more likely to be scheduled to run. Changing priority levels on run-time is possible however it is discouraged [6]. Instead the user (programmer) is supposed to guarantee the tasks' deadline by analyzing the schedulability.

Pro's:

- The task with the highest priority is always run which gives predictability
- Lots of priority levels gives the user control over the scheduling

Con's:

- Starvation is possible because there are no guarantees that a process with a low priority level ever will be run

2.3 Why VxWorks is hard real-time

The scheduler in VxWorks does not implement any form of admission control, meaning that it does have any run-time protection against impossible scheduling. Why is it still a hard real-time operating system? Because it's predictable and deterministic. It is the user's responsibility to analyze the scheduling (which can be done offline with timing tools). The goal with the VxWorks scheduler is to provide the user with predictability and not most optimal algorithm.

2.4 Priority inversion

Priority inversion is the name of the problem which can occur with preemptive priority-based scheduling algorithms. When a lower priority task is preempted by a higher priority task it still holds the resource it was using. If the higher priority task needs the same resource it will have to wait for the lower priority task to finish. While the lower priority task is finishing a new task with medium priority preempts the lower priority task. This makes the higher priority task wait for a lower (medium) priority task which is not fair in a priority-based algorithm. To solve this VxWorks has implemented a priority-inheritance protocol. Before the lower priority task is run (to finish and free the resource) it's priority is temporarily changed to the same priority as the higher priority task (the priority is inherited). This prevents the medium priority task to preempt the lower priority task and when the lower priority task has freed the resource its original priority is set and the higher priority task preempts. Since the resource now is free it can run. [4]

2.5 Summary

VxWorks is a widely used commercial real time operating system. The scheduling algorithms provided in VxWorks are round-robin and preemptive priority-based. Used together they make up a good predictable scheduler leaving the user with a great deal of control. The priority inversion problem is solved by implementing a priority inheritance protocol.

3 Comparison

In this chapter we will discuss the differences between, and compare, the scheduler in the two real time operating systems. The chapter is divided in two parts. In the first part we compare the algorithms based on a theoretical view. In the second part we compare the algorithms based on a contextual view. In what context are the algorithms used and how well do the algorithms fit in that environment?

3.1 Theoretical view

Since both RTLinux and VxWorks has implemented the POSIX scheduler it is possible to make the operating systems behave in the very same way regarding scheduling. The default scheduler in VxWorks is very similar to the POSIX Pthread scheduler. In fact we find it a bit odd that VxWorks implemented POSIX scheduler at all, or the opposite question, why they bothered to write their own scheduler. The answer to this might be that VxWorks was implemented before the POSIX.1b standard (with the real time extensions) was released. When the POSIX.1b was later released they probably wanted to add the support for compatibility reasons. While VxWorks' default scheduler is similar to the POSIX scheduler, the default scheduler in RTLinux differs a lot more. With the two more advanced (compared to round-robin and first-in first-out) scheduling algorithms earliest-deadline-first and rate-monotonic the RTLinux user is allowed more freedom while choosing scheduling algorithm.

3.2 Contextual view

Both VxWorks and RTLinux give users the possibility to choose between different scheduling algorithms depending on what's needed, although in VxWorks the user can in fact only chose between different approach on how to handle tasks with the same priority. Either use a non-preemptive first-come first-served or a round-robin approach. RTLinux provides a more varied range of algorithms to use in different situations. The contexts in which these two real time operating systems are used can probably be assumed to be the same - embedded systems where the response time to certain events is crucial and the task switching latency needs to be minimal. Another key factor for the behavior of an real time operating system, possibly the most important one, is that it is predictable.

The task switching latency is indirectly affected by the choice of scheduling algorithm, or at least the number of task switches that is done. Assume we need to minimize the amount of context switches of the CPU done by the scheduler (maybe the hardware switching latency is high), which one of these algorithms is the best suited? In the case of VxWorks the round-robin approach could produce a lot of context switching. Increasing the size of the time slice (which is set by the user) will reduce the number of context switches, but disabling the round-robin option totally would be even better. The preemptive priority-based algorithm would in this case handle tasks with the same priority on a first-come first-served basis resulting in a minimal number of context switches. Considering RTLinux instead, which of the possible scheduling algorithms produces the least context switching? With the available POSIX scheduler the use of first-come first-served will produce less context switching than the round-robin approach, with the same motivation as for VxWorks above. The rate-monotonic algorithm is a good way of keeping the context switching low because of the design of the algorithm. The earliest-deadline-first algorithm on the other hand will produce more context switching because of the dynamic changing of priorities.

Another important issue is how predictable and deterministic the operating system is. For a real-time operating system to be hard it does not have to guarantee that no deadlines are missed but to guarantee that the system is deterministic and leave the deadline-guarantee to the software. The scheduling algorithms used, directly affects the predictability of the system. What can be said about predictability of the algorithms used in RTLinux? A common misconception is that the rate-monotonic algorithm is more predictable than the earliest-deadline-first algorithm in an overload situation but this is not true [7].

The third issue we want to address here is the interrupt latency, which of course is very important that it is kept very low. The interrupt latency is affected whenever the system has to disable interrupt. This is done for example when the scheduler is switching between tasks. A badly designed scheduler with a long search time to find the next task will increase the interrupt latency if an interrupt occurs during that time. To minimize the effect of this problem different ways to structure the ready queue can be used. For example when the scheduling algorithm used is preemptive priority-based a ready queue sorted by priority is effective because the task with the highest priority level will be found in "no" time. With an algorithm such as earliest-deadline-first (EDF) the priorities are constantly changed by the scheduler to match the deadlines approaching resulting in a ready queue which has to be resorted all the time (if the ready queue is sorted on priority). This resorting has to be done more often the more tasks the system has to handle, and this will probably affect the overall performance.

RTLinux has it's default scheduling algorithm which uses fixed periods for each task and preempts each task when the time slice is over, likewise VxWorks has it's POSIX FIFO scheduler, both of them not requiring very much overhead for context switching. Still the POSIX FIFO scheduler lacks protection against starvation and the default RTLinux scheduler lacks protection against impossible schedules, making both of them hard, if not impossible, to use in critical environments.

4 Conclusions

In this chapter we discuss the result of the comparisons made in the previous chapter and come up with conclusions on when RTLinux is appropriate to use, and when to choose VxWorks. We also discuss whether one of the real-time operating systems can be said being better than the other.

RTLinux seems to have more advanced scheduling algorithms available than VxWorks, some of them thanks to it's user community[3] and the fact that it's module based and open source. VxWorks on the other hand doesn't aim to serve the user with algorithms to do the work of putting up a working schedule, but hands over this job to the programmer. This way VxWorks does not have the same problem with delay in context switches. But the fact that RTLinux offers both simpler, yet fast, algorithms and more advanced choices makes it competitive to VxWorks.

For low cost systems running in many different environments and contexts where you may be in need of different algorithms depending on where the system is being used, RTLinux will probably be a good choice since it has a wide set of scheduling algorithms, all shaped to fit in different situations. They are also easily changed during run time due to the Linux and RTCore modularity.

VxWorks on the other hand seems to be the best choice if you are developing a system which you want a hundred percent control of, making all decisions yourself and leaving nothing at random. Predictability is a strength that VxWorks has, which makes it hard to beat in tougher environments, and even though it's predictable it does not suffers from problems with frequent context switching as with for example RTLinux' EDF algorithm.

To summarize, it's hard to say that RTLinux definitely is a better choice than VxWorks, or vice versa. Everything depends on the situation, what the purpose of the system is, what environments it will run in and how much work one is willing to put into it. VxWorks seems to require more work to grow into it's full capabilities, where as the user might not even be able to get that much out of a RTLinux system. Though RTLinux seems easier to get going and become useful.

References

[1] http://www.rtlinux.com

[2] http://www.linuxfocus.org/English/May1998/article44.html

[3] http://www.linuxdevices.com/articles/AT3694406595.html

[4] Operating System Concepts - Silberschatz, Galvin, Gagne (Seventh edition)

[5] http://www.windriver.com/vxworks/

[6] http://www.cs.huji.ac.il/labs/danss/presentations/realtimeos.ppt

[7] http://www.cas.mcmaster.ca/~downd/rtsj05-rmedf.pdf