Comparison of scheduling in RTLinux and QNX

Andreas Lindqvist, andli299@student.liu.se
Tommy Persson, tompe015@student.liu.se

19 November 2006

Abstract

The purpose of this report was to learn more about scheduling, and how it's done in the two Real Time Operating Systems QNX and RTLinux.

The report contains our findings about scheduling with focus on the different scheduling algorithms in the two RTOS:s. These algorithms include the rate-monotonic algorithm (RMA) and the earliest-deadline-first algorithm (EDF) in RTLinux and FIFO and Round Robin as well as sporadic scheduling under QNX.

Our main conclusion is that both OS'es would be sufficient for meeting the demands of a hard real-time operating system. However, we need to perform a task analysis before configuring and test the system, that should host our real time application.

Chapter 1

Introduction

1.1 Threads and their states

A thread can be in different states. The states are running, blocked, and ready. On a single process system only one thread can have the state running. The ready state is assigned to a thread waiting for the CPU. The block state occurs when a thread is sleeping, waiting for a mutex owned by another thread, or waiting for a message from another thread. When creating a thread, a TCB, thread control block, is created and is related to the thread. The TCB stores information about the thread. That can for example be the state, priority and which scheduling algorithm to use.

1.2 Scheduling and its performance

Scheduling is the kernel process where the operating system chooses which thread to run from the ready queue. The dispatcher allocates the CPU to the thread the scheduler selects. CPU scheduling is the foundation for concurrent programming. Switching the CPU between threads makes the operating system more effective.

Pre-emptive scheduling takes place when the scheduler interrupts a running thread and allocates the CPU to another thread. Non pre-emptive scheduling occurs when a running thread switch state to blocked, or terminates.

There are different algorithms the scheduler can use to decide in how and in which order the threads are switched between states.

There are different scheduling criteria. One is CPU utilization. The CPU utilization can vary from 0-100%. We want to keep the CPU on a computer as busy as possible. Another criteria is the waiting time, the time a process waits in the ready queue.

We must have methods to compare different systems' performance, the most reliable method is to test algorithms on the actual system and monitor its performance in the real world.

Chapter 2

Real Time Operating Systems

A Real Time Operating System, RTOS, is valued for how quickly and predictably it can compute a problem. To realize a RTOS it needs to have pre-emptive priority based scheduling, priority inheritance and a pre-emptive kernel.

In real time systems determinism (the worst-case transaction rate) and responsiveness (the transaction rate) are critical goals. Before choosing a RTOS we must make clear our application's performance demands. We must know the tasks, their deadlines, periods and compute time. [11] The information is necessary to perform a detailed task analysis, so we can configure and test the system that should host our real time application.

2.1 QNX

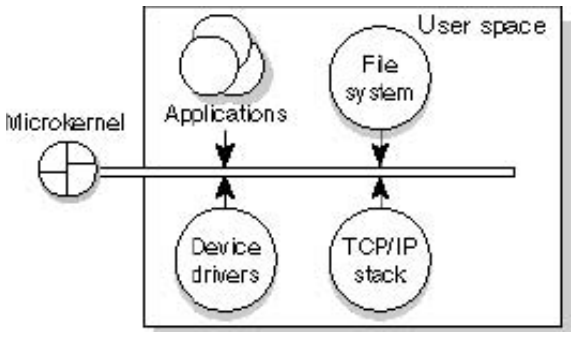

Figure 2.1: QNX's microkernel design.[10]

QNX (pronounces queue nicks) is a proprietary non-open source microkernel-based hard real-time operating system.

The OS first shipped in 1981. (It started with two students programming a RTOS they called "Quick UNIX". AT&T wasn't happy with that name so it was changed to QNX.) The latest QNX RTOS is Neutrino v.6.3 and it is produced by QNX Software Systems with headquarter in Ottawa, Ontario, Canada.

2.1.1 Main features in QNX

QNX has a microkernel with thread scheduling, message passing and real time services. Other operating system services are called resource managers and are placed in user space. They are optional components and can be plugged in/out in run time. Hence QNX has device drivers and file system in user space. If any part in user space fails, it can be restarted. QNX supports running on many architectures from everything from embedded systems to SMP systems. Although QNX can be used on networked, high end SMP machines with GBs of physical RAM, nevertheless the microkernel can be scaled down to a very small size. (The QNX 4.x microkernel is only 12 kb.)

QNX is POSIX 1b Real-Time extensions compliant, with FIFO and Round Robin priority based scheduling. QNX does not only use priority based scheduling, it also provides priority based message passing.

As for a good anecdote, rumour (http://www.lysator.liu.se/upplysning/971015.html) says that in former Soviet Union states QNX was very popular, because of providing a multi-user real- time OS with GUI, capable of running on a 286 with 1MB RAM. Maybe it wasn't so unpopular among the computer club Lysator's members at LiU either, because of the high presence at the lecture.

2.1.2 SMP

QNX supports SMP scheduling. The standard way is to put the threads in a common queue and take the highest priority thread and run it on an available processor. The other way is called processor affinity. In the TCB for each thread you put an affinity mask telling the scheduler which CPU to schedule the thread for. There is a bit mask for every processor. The affinity mask can be set by the thread itself by an affinity mask system call.

2.1.3 Priority levels

To be Real-Time POSIX compliant the OS has to have a minimum of 32 priority levels. QNX offers 64 priority levels. The priority-based pre-emptive scheduling in QNX is a major key for giving time-critical tasks the possibility to be executed in the right time. The priorities range from 0 to 63, with 63 as the highest priority. The special thread idle runs at priority 0 and is always ready to run. Each priority level has its own ready queue. A thread who wants to run is put in the back in the ready queue for that priority level. The thread in the head of the queue with the highest priority will get the CPU. Each thread has both a priority level and a scheduling algorithm in its TCB. Hence the scheduling algorithm is used on a per thread basis. The scheduling algorithm is inherited by the parent and it is possible for the thread itself to change the scheduling algorithm in run time.

The scheduling algorithm only matters when there are two threads with the same priority level in the ready queue. A higher priority thread always pre-empts a lower priority thread POSIX, the Portable Operating System Interface for uniX, is a system application program interface. The POSIX 1b is the standard's extension for real time systems, which QNX complies with.

PThread scheduling is the POSIX way of scheduling real time threads. Pthreads consists of FIFO and RR (Round Robin). In FIFO there is no time slicing between threads of the same priority. The highest priority thread in the head of the ready queue is granted the CPU until it is finishes or yields. RR is FIFO with time slicing between threads of the same priority. In Pthreads you can get, or change, the scheduling algorithm for a thread.

QNX supports FIFO scheduling, Round Robin scheduling and Sporadic Scheduling.

2.1.4 Priority based scheduling with FIFO

In real time systems it is very important to serve the requesting threads with the CPU resource. That is why the priority based scheduling is a key solution. A priority is assigned to each thread. A thread with higher priority than another thread is allocated to the CPU first. Higher priority tasks always execute before lower priority tasks There are pre-emptive and non pre-emptive priority based scheduling. Pre-emptive is when a thread with higher priority than the running process arrives to the ready queue. The scheduler puts the running thread to ready state and the arriving thread is given access to the CPU. The state for the thread that pre-empts the other is now changed from ready to running state.

The first-in, first-out scheduling algorithm is absolutely the easiest scheduling algorithm. It is implemented using a queue. It is used when two threads have the same priority. Process A and B have the same priority. The CPU is busy running a thread with higher priority. The running process is not interrupted with FIFO, so the CPU keeps processing the running thread until the thread yields the CPU. Then it's a simple matter of keeping track of in which order the two processes A and B arrived to the ready queue. The thread that came first in is first out. The draw back in FIFO scheduling is that a process has to wait until a possible longer process has finished. With FIFO we certainly don't want a process doesn't behaving and hogging the CPU.

A problem with priority based scheduling is starvation. A thread with low priority arrives to the ready queue, but the scheduler allocates the CPU to threads with higher priority and the waiting time for the thread grows and grows. There is said that when they shut down the IBM 7094 at MIT in 1973, there was a low priority process in the ready queue that had been in the queue since 1967 and had not been given CPU-time to execute and finish its job. The solution to the starvation problem is known as aging. The threads in the ready queue are given higher and higher priority as their waiting time increases; finally their priority is high enough for the opportunity to access the CPU resource.

In QNX, a thread using the FIFO scheduling algorithm and is running keeps running until it blocks (voluntary releases the CPU), or is pre-empted by a thread in the ready queue with higher priority.

2.1.5 Priority based scheduling with Round Robin

RR scheduling is developed with time slicing systems in mind. Apart from FIFO functionality the RR scheduling algorithm implements pre-emption to switch between the processes. RR is used when two or more processes have the same priority. A time quantum is defined. The ready queue is seen as a circular queue. The scheduler goes round the queue and gives a process the chance to use the CPU for up to one time quantum. If the process has a shorter CPU burst than the time quantum the process yields the CPU voluntary. The CPU scheduler picks the thread in turn from the ready queue, sets a timer and switch to next thread in turn after the time quantum has been reached. The process that just got switched is put at the tail of the ready queue. With n equal priority level processes and time quanta q a process get $1/n$ of the CPU time. It has to wait $(n-1) \times q$ time units until next CPU allocation. With RR the average waiting time is often long. The challenge with Round Robin scheduling is to select a good time quantum, which is the greatest contributor to the performance of the algorithm. Too large time quantum makes the RR function as FIFO, while a too small time quantum makes the algorithm ineffective, by the many context switches the dispatcher has to make.

In QNX the Round Robin scheduling algorithm uses time slices. RR's functionality is the same as FIFO's except this feature. A thread keeps running until it blocks, is pre-empted or has consumed its time slice. A time slice is the time a thread can use the CPU. When the thread has used its time slice it is put at the end in the ready queue and the thread at the head of the queue will use the CPU for up to one time slice.

2.1.6 Sporadic scheduling

Sporadic Scheduling is used when we want to prescribe a time limit a thread can execute within a period of time. The drawback with sporadic scheduling is that the algorithm produces overhead. The algorithm can be used for not periodic events which need service. With sporadic scheduling a threads priority can oscillate between two priority levels. A budget is used, i.e. the maximum time within a period of time the thread can execute on a higher priority level. The replenishment period is the time period in which the thread can use its budget. When a thread exhausts its budget the priority drops until the budget is replenished.

2.1.7 Message passing using priority inheritance

QNX provides priority inheritance. The QNX server process receives messages from threads in the system. The server process spawns a thread for each request and the new thread's priority is set to the same level as the requesting thread. By this way higher priority threads get its requests served before lower priority threads. Also when a higher priority thread sends a message to a lower priority thread we want to ensure that the lower priority thread gets its job done. Priority inheritance happens when the receiving thread inheritances the priority of the sender so it can complete its job in time.

QNX uses a non-POSIX message passing system within the kernel, the Neutrino message-passing services. The service provider executes the highest priority client's request at the same priority level as the client, and from a scheduling analysis perspective on the execution time the client and the server can be treated as only one thread. While POSIX uses queues, the QNX message passing system copies the message directly to the receiving threads address space, using priority inheritance. The speed of the message passing system is therefore limited only by hardware speed.

2.2 RTLinux

RTLinux is a miniature real-time operating system that operates a level above the standard Linux kernel. This front-end works by running the Linux kernel as a separate task among other real-time processes as well as disabling the Linux kernels ability to disable interrupts. Any interrupts received by the RTLinux mini-kernel is either sent to the Linux kernel or the other real-time tasks. [1]

RTLinux is used in many different embedded applications, from jet engine tests and robotics to network appliances. [4]

The RTLinux system is built in a modular fashion so it is relatively simple to write your own schedulers or memory managers. In this report however, we will stick to the three included schedulers. Two of which will be given a more in-depth look.

Apart from the simple preemptive fixed priority scheduler that is the default in RTLinux, two other scheduler packages are included in the standard RTLinux distribution. They are the Rate Monotonic (RM) and Earliest Deadline First (EDF) algorithms. [3]

2.2.1 Rate-Monotonic Scheduling

In a system using the rate-monotonic each periodic process entering the system gets a priority based upon the length of its period. The shorter period the process has, the higher priority it is given. The rate-monotonic algorithm then schedules these periodic processes according to their priority and if a new, higher prioritized, process enters the systems it may preempt any other running task.

We can consider two tasks t_1 and t_2 in a system using the rate-monotonic algorithm. t_1 has a period of $T_1 = 50$ ms and t_2 has a period of $T_2 = 100$ ms. The two tasks have a worst-case execution time of $C_1 = 25$ ms and $C_2 = 40$ ms. the RMA will assigned the highest priority to t_1 due to it's shorter period. During the first period, $t1$ will finish without interruptions. t_2 however won't be able to finish before it's time for t_1 to execute again. When t_1 once again finishes t_2 is allowed to resumes and finishes within it's period starts again. This process repeats ad infinitum.

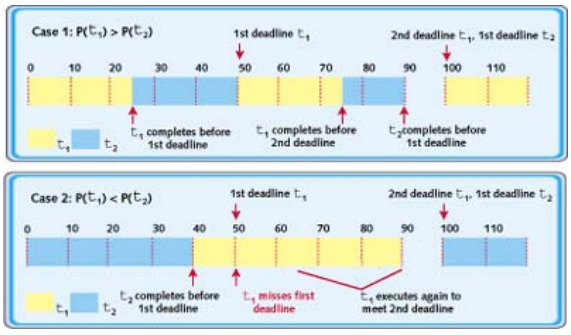

Figure 2.2: This figure[2] illustrates the above example as well as what could happen when priorities are not assigned properly.

Among other static priority scheduling algorithms, the rate-monotonic is considered optimal because if a set of tasks can't be scheduled, neither can any other of the static priority algorithms. [1]

Fixed priority schedulers like the RMA is simple to implement, has a low overhead and overloads like the EDF might suffer are a non-issue since all tasks are periodic. This makes the rate-monotonic algorithm suitable for hard real-time operating systems.

One of the rate-monotonic algorithms weaknesses is that it is conservative and may not fully utilize the CPU under all circumstances. In fact according to the formula (2.1) for the worst-case utilization of n processes, when n grows bigger, only 69% is used [1], though the average workload comes closer to $\approx 88\% - 92\%$ [5].

$$W = n(2^{\frac{1}{n}} - 1) \tag{2.1}$$

W can also be used to check if a given set of tasks is schedulable, if the CPU utilization presented by the task set is below W then the tasks are schedulable, otherwise, if the utilization is above W, the set *may* be unschedulable. [2]

A way to achieve 100% utilization is to use harmonic periods. Harmonic in this context means that each task is a multiple of all tasks that has shorter periods.

2.2.2 Earliest-Deadline-First Scheduling

Unlike the RMA, the earliest-deadline-first algorithm is an algorithm that assigns priorities dynamically. The priority a process receives is related to it's deadline, the earlier deadline compared to other processes, the higher priority it receives. The earliest-deadline-first algorithm is preemptive.

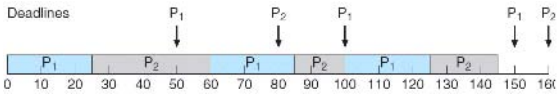

Figure 2.3: This figure [5] shows how the EDF-algorithm works.

An illustrative example [5] is the one where the task t_1 has a period of $T_1 = 50$ ms and a worst-case execution time of $C_2 = 25$ ms and where the task t_2 has an execution time of $C_2 = 35$ and a period $T_2 = 80$. At time index 0 t_1 has the highest priority since it's deadline is the closest one of the two. t_2 starts executing after t_1 has completed and continues executing even when $t1$ comes again, since t_1's deadline now is after the one of t_2. At time index 85, t_1 has finished executing within it's deadline and t_2 begins it second period. t_2 is interrupted at time index 100 when t_1 yet again begins a execution cycle. When t_1 finishes, t_2 resumes and both tasks finishes within their respective deadlines.

The EDF algorithm also doesn't require the processes to be periodic or a constant execution time like the RMA does. All it requires is that when it starts running, it also supplies the scheduler with its deadline. The algorithm can also be preferable due to it being able to schedule any set of tasks as long as their total CPU utilization is at or below 100%. Theoretically that is, on real systems interrupt handling and context switching may be too costly to achieve a full 100% utilization.

Earliest-deadline-first, being a dynamic priority scheduler is more prone to overloads than a fixed priority scheduler like RMA and offering only statistical guarantees, might be more suitable for soft real-time systems.

Chapter 3

Conclusions

The hard RTOS QNX has scheduling algorithms on a per thread basis. Not only can the scheduling algorithm for a thread be changed in run time, each thread's priority can also be changed.

QNX supports FIFO and Round Robin scheduling for equal priority threads. Furthermore, QNX supports sporadic scheduling giving the possibility to budget the time a thread is running at high priority.

Although the microkernel structure, which usually has higher overhead than monolithic kernels like Linux, the design in QNX is addressing the issue and the Neutrino message passing system gives the system speed and efficiency qualities.

RTLinux supports the fixed priority rate-monotonic algorithm as well as the earliest-deadline-first algorithm.

Depending on the scheduling algorithm chosen, RTLinux is fit for hard real-time. It can of course also be used as a soft real-time operating system. The recommended choice when developing systems with critical functionality is the rate-monotonic (RM) algorithm due to the EDF-algorithm's overloading issues and lack of any real guarantees.

Before choosing a RTOS we must make clear our application's performance demands. We must know the tasks, their deadlines, periods and compute time, then it's time for a detailed task analysis before configuring and test each system with our real time application.

Bibliography

[1] Silberschatz, Galvin, Gagne, *Operating System Concepts*, Seventh Edition.

[2] Stewart, David and Michael Barr. *Rate Monotonic Scheduling*, Embedded Systems Programming, March 2002, pp. 79-80

[3] Yodaiken, Victor. *The RTLinux Manifesto*

[4] FSMLabs http://www.fsmlabs.com

[5] Kessler, Christoph and Hansson, Jörgen. *Real-Time Operating Systems* (lecture notes) http://www.ida.liu.se/ TDDB72/slides/2006/RTOS-2006.pdf

[6] IEEE POSIX®Certification Authority. http://standards.ieee.org/regauth/posix/index.html

[7] Jane W. Liu: *Real-time systems*

[8] Silberschatz, Galvin and Gagne. *Applied Operating Systems Concepts*

[9] QNX company website. http://www.qnx.com

[10] *QNX Neutrino System Architecture* and *QNX Neutrino Programmer's Guide* website. http://www.qnx.com/developers/docs/momentics621_docs/momentics/index.html

[11] Bill O. Gallmeister *POSIX.4 Programming for the real world*