Advanced Tools

REAL-TIME PROGRAMMING

October 17, 2017

Arthur Heimbrecht
Matrikelnummer: 3159717
Universität Heidelberg
Fakultät für Physik

03/07/2017

Contents

Introduction

As of today, many systems and applications become more and more seamlessly integrated into our world. Ideally these systems work in the background without us even noticing them despite their enormous impact to our every day lives. These systems rely heavily on a balance of available and requested computational power. This is of special importance on small scale systems, which are often called embedded systems, where the available computational power barely exceeds the need of its applications to make the systems as cheap and small scale as possible.

To allow for this, the programming of such devices becomes all the more important as resources need to be managed wisely while important tasks need to be executed all the time. To make this even more interesting, these systems are often subject to a so-called 'real-time constraint'. This is equivalent to a series of deadlines which should be met by the system. Otherwise the consequences could be fatal.

Because this time-dependence requires systems not to work on their own time scale but in **real time** (also called 'external time'), the terms 'real-time programming' or 'real-time computing' are used. To guarantee meeting these deadlines, real-time systems for example need to prioritize time-relevant tasks while other tasks may be delayed, all the while assuring a correct output at the deadline. This should not be misunderstood as the system being fast - although it often goes hand in hand with this. 'Real time' rather means that at a specific point in time there must be a reliable result. The results are also described as **deterministic**, which means that there must not be a case which the system can not handle or which produces an unknown output [2].

Systems, which make use of real time computing go back to the beginnings of the digital era. Back then it was already important to categorize certain tasks such as input signals due to their importance to the system and give them a higher priority. Other real time systems would be airbags or anti-lock breaks. Both need to analyze multiple inputs in a fraction of a second and react accordingly in order to prevent possibly fatal outcomes.

It is clear that real-time constraints can be seen almost everywhere and they will become even more important in the future. It is therefore worth taking a look into this topic during the course of this report.

It will start with different constraints to real-time systems as well as different systems that house real time applications. It continues with a multitude of tasks and ways to schedule these in order to meet various deadlines. Next it will emphasize on available resources and problems that can emerge in real time applications. At last it will conclude

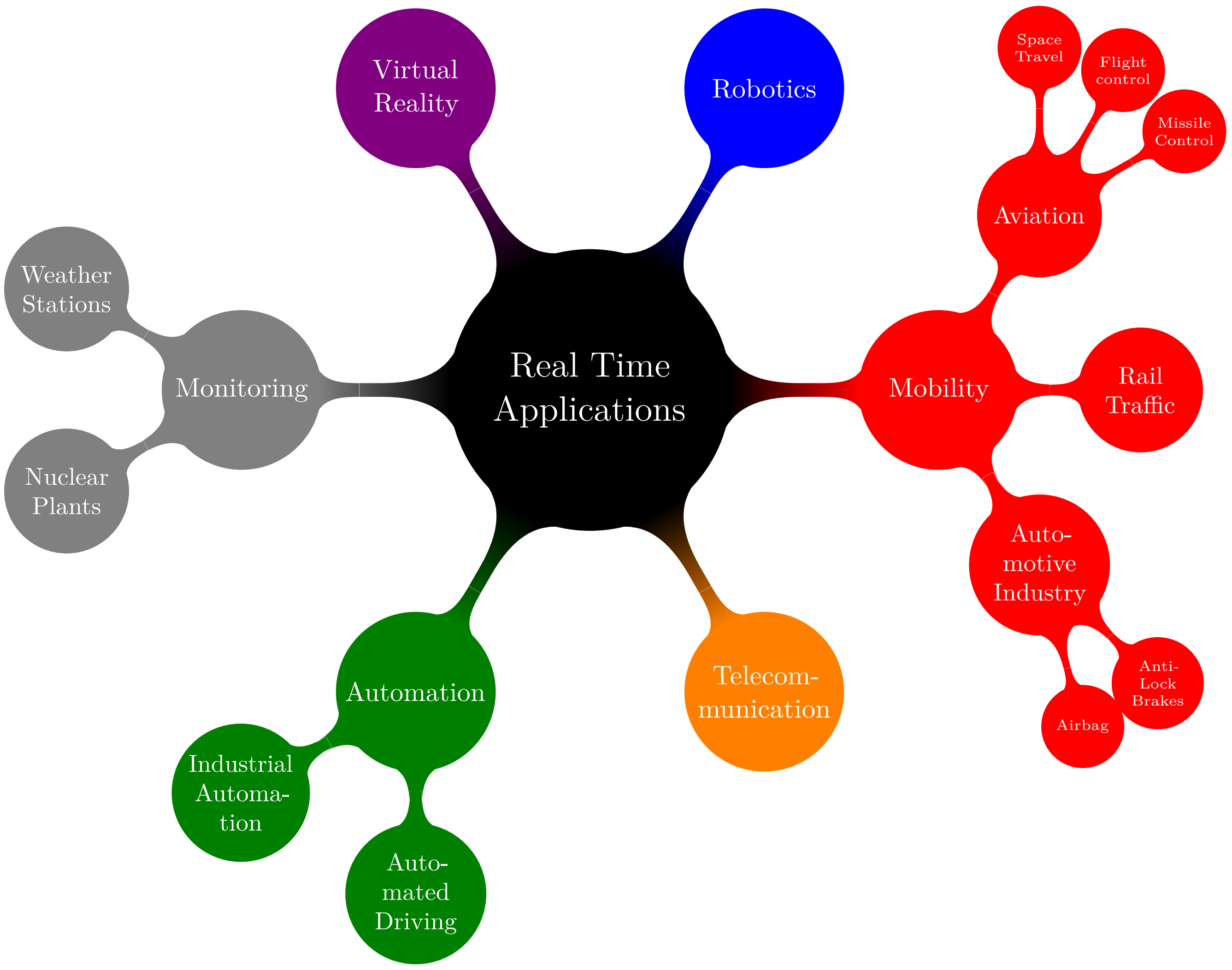


Figure 1: Sectors and Examples with Real Time Applications

with a short summary of key aspects in real time programming and also an outlook to the possibilities of developing an real time application. Ultimately the appendix gives some further interesting information.

http://www.elec-tro.fisica.unlp.edu ter1.pdf.

Chapter 1

Differences in Real Time

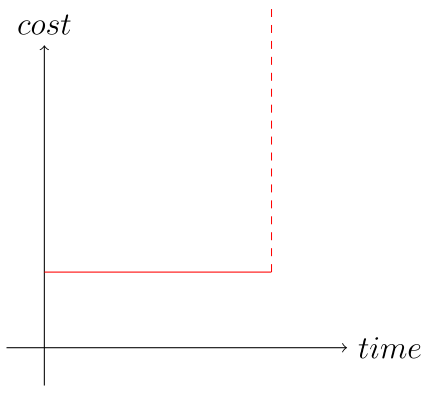
There is a multitude of real time applications that vary in their constraints as well as their implementation. Nonetheless there is a group of real time applications that is often used as the prime example for real-time n general which is hard real time [11].

In **hard real time** a deadline must be met, otherwise the program will fail. This can be visualized by a cost function which is constant for any time before the deadline and then becomes infinite at the deadline (figure 1.1a). A simple example would be that of an airbag:

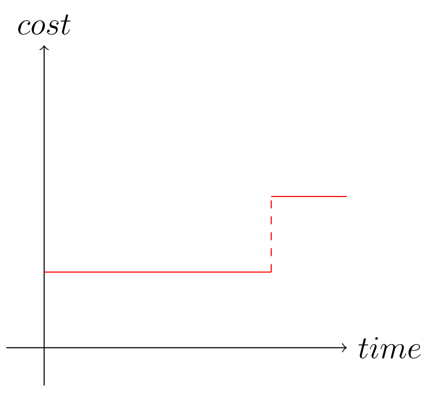
Whenever a crash happens, the airbag's electronics analyze the signals of various sensors inside the car for whether an actual crash has happened - or rather is happening - or if someone just backed up into a hydrant. In the case of an actual crash this decision has to be made in a tenth of a second since it would be worthless after the point in time when the airbag would have helped. Of course this is a drastic example but it illustrates the problems that might come up when working with hard real time. How can one guarantee to meet the deadline? What should happen in case of the 'unexpected' or how can it be avoided? What if the system does not meet the deadline? In any case missing the deadline is generally perceived as a failure of the program and should be handled accordingly.

Still there are other applications where not making the deadline will only become problematic if it happens repeatedly. This is called **soft real time** and a good example is network communication:

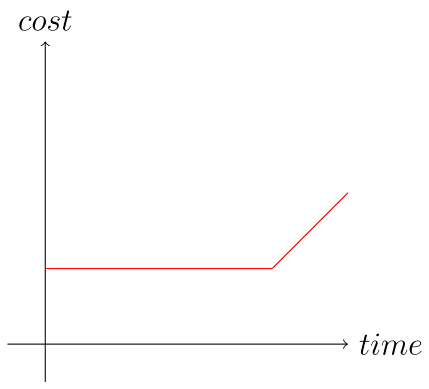
If there happens to be a time-out in the midst of a file transfer this is not a big deal, as the process of transmitting the package can be stopped and the package can be sent again. This happens typically in any larger network and if it happens once every few hundred/thousand packages it will not influence transmission rates significantly. But still it might become a problem if it happens too often and the overall transmission rate drops



(a) hard real time



(b) soft real time



(c) firm real time

Figure 1.1: Cost functions for different classifications of real time

or transmission is stopped altogether. A cost function will look similar to that of hard real time but with finite cost after the deadline (figure 1.1b).

Closely related to this is **firm real time**, where the process is not stopped but kept alive and the result can still be used if the delay is small enough. Video playback gives us a good example for this:

If the system cannot render a frame in time, the frame can be delayed a little instead of skipping it completely (this would be soft real time). In this case it would not affect the viewing experience much. But the more the frame is delayed the worse it will be. As a cost function this is represented by increasing the cost over time when the deadline is passed (figure 1.1c).

Between these general types of real time there are no clear boundaries and terms like 'weak hard real time' do not differ much from these basic concepts.

The same goes for another classification of real-time systems. Looking at the examples we already used, there is another difference between an airbag and the playback example.

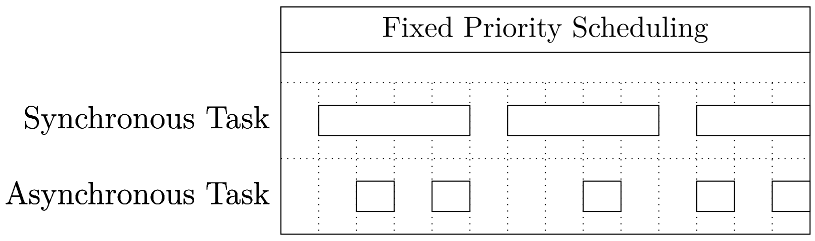


Figure 1.2: Gant diagram for synchronous and asynchronous tasks

One is started by an event at any time thus it is an **event-driven** or **asynchronous task** the other is started periodically and thus it is an **periodic** or **synchronous task**. The terms synchronous and asynchronous originate from the principle that basically every system has an update cycle or a periodic time frame in which it is able to react. RTsystems

When thinking a little more about the concepts of firm real time and soft real time, it becomes clear that basically all applications that we use would fit into some concept of real-time. If a personal computer misses the deadline of the users patience too often, the system might be replaced soon enough. If a simulation takes decades to finish the result might not be worth anything when it finally becomes available. So it might be worth looking into some general rules of real time programming even though your system might not classify as real-time at first.

Chapter 2

Embedded and General Purpose Systems

2.1 Classification

As already mentioned in the introduction, (real time) programming only becomes a challenge when developing for systems that have a limited amount of resources or just barely enough to handle the task for which they were designed for. Normally these systems then are embedded into a larger scale system where they have a dedicated purpose and which makes them **embedded systems**. This kind of systems accounts for an incredible amount of up to 98% of all produced processors [9].

The remaining 2% are mostly used in **general purpose** machines such as personal computers. It is also possible to create real-time programs for these although it is often perceived as a minor challenge, regarding the computational power of these systems. But meeting distinct deadlines reliably can be even more challenging on a GP system than on an embedded system.

On the other side, in order to meet the low specifications of embedded systems an ideal program should be written in assembly, which is the lowest human-readable form of programming as this issues instructions directly to the processor. Still, it is obvious that this kind of programming can become very tedious, hard to understand as well as maintain. Thankfully there is the alternative of programming in C or any other high-level language. The advantage is that programs are easy to read while the compiler will generate the assembly code from the source code. But does this still allow for timeliness as well as determinism in a program? Using the right tools and techniques it does!

2.2 Basic Example

A first example of a program which will be the basis of this report is the `LED_simplewhile.c` program (see 6.3 for more information).

A Raspberry Pi 3 Model B is used as a model system. Although it qualifies as a general purpose system, its relatively low specifications can easily be exploited later on, when showing some techniques for real-time programming. There are also only a few system processes in the background that might interfere and the GPIO ports allow for an easy interaction with the environment.

The first example is used to explain basic thoughts behind real time programming. The purpose of this program is to simply toggle an LED on and off repeatedly. This can either be done to save some power compared to having the LED be on constantly or to continuously signal the state of a system. If put in a forever-loop, the system will repeat this task as often as possible in a given time.

But this raises some problems: We do not know at which frequency the loop cycles and most likely it will be much faster than our eyes can see (therefore the light seems steady) and/or faster than the sensor data is updated which would waste valuable resources. In any way we do not profit from the loop running at a higher frequency which also wastes power that might be needed for something else. To tackle this problem we will introduce **interrupts** and **clocks** (section 3.1).

In case the loop does not run that fast it might encounter a different problem for additional tasks in the loop. If these tasks need too much computing time the loop will slow down which we may notice as a flickering light. Also different tasks could be combined that have different periodicities (remember synchronous tasks) and one must 'arrange' or schedule them. This problem will we handled by **task schedulers** (section 3.2).

On general purpose systems we do not know what other tasks are running and if they clobber up the memory for our task it might be 'swapped out'. Alternatively some data may not be in the memory altogether but is 'cashed'. This is a problem that concerns mostly general purpose systems but in any case it requires to **manage memory** (chapter 4) wisely not only if there are multiple tasks running on our system.

Regarding real-time programming these problems can concern embedded systems as much as general purpose ones and therefore this report will focus more on general purpose systems for simple examples and demonstration purposes. Once again, one should keep the very limited computational power in mind when it comes to embedded systems. This valuable resource should be harvested as much as possible. Using larger variable sizes than needed and declaring constants not constants slows down the program and thus should be avoided in general.

Chapter 3

Multitasking

To start off with multitasking it is necessary to introduce clocks and timers first. A simple example that resembles the previous example would be a temperature sensor that constantly measures the temperature of a system inside a forever loop that checks if this temperature is critical. If so the system should be shut down to prevent any heat damage.

If this was to be implemented similarly to the `LED_interrupt.c` example there would be two kinds of problems. One would be wasting resources if the loop's frequency greatly exceeds the sampling rate of the sensor. The other would emerge even as the update frequency barely exceeds that of the sensor. In this case the system will at times be busy analyzing an old value a second time although a new value becomes available just as the loop started over - this effectively wastes a cycle. This cycle could possibly be critical to preventing heat damage and therefore one wants to **synchronize** the loop to the sensors sampling rate.

3.1 Clocks and Timers

To do so one can add a timer at the end of the loop and halt the loop until the timer has finished. For example the loop might wait that 100 microseconds have passed since the last cycle and then start again. There exist standard system timers for this. Although these typically come with a disadvantage.

Since time is stored as a value of seconds since 0:00 am January 1st 1970 one can encounter a problem regarding accuracy. A system therefore offers time in nanoseconds as well - an accuracy most systems simply cannot achieve or do not need. The problem which emerges from this becomes obvious if the system clock's tick is not an integer multitude of nanoseconds (one 1024th of a second for instance). In this case the clock rounds to the next complete integer in nanoseconds. This gives an error of up to 0.5ns per tick. To compensate for this, the clock will simply increase/decrease the last tick of every second to accommodate for this deviation. Which can result in problems if the time measurement is in the order of system ticks and it happens to hit the last tick of a second.

$$\frac{1}{1024}\text{s} = 967563.5\text{ns} \xrightarrow{\text{rounded}} 967563\text{ns} = T_{tick} \tag{3.1}$$

$$\xrightarrow{\text{in one second}} T_{tick} * 1024 = 990784512\text{ns} = 1\text{s} - 9215488\text{ns} \approx 1\text{s} - 9.5 * T_{tick} \tag{3.2}$$

Fortunately there are libraries that offer clocks that are not prone to this error. One of these is the POSIX library. This library is often used for real-time applications and offers a multitude of tools (this will be further discussed in 6.2). POSIX' `nanosleep` specifically rounds up to the nearest integer in nanoseconds which should be kept in mind for highly time-dependent applications but avoids the increased tick error by doing so.

This allows for expanding the `LED_interrupt.c` example with `nanosleep` as in `LED_interrupt_nanosleep.c`. Now the blinking lights will blink with (almost) exactly 60Hz which is barely visible for the human eye.

3.2 Schedulers

This example can also be extended to multiple lights. It started with 60Hz, which should show next to no flickering. Though if one looks closely it might be visible. Now a second LED is introduced flickering at 30Hz , which should really show obvious flickering. Adding a third LED, that flickers at 15Hz, one can well observe 'slow' flickering.

All of this is fairly easy to implement as all frequencies are harmonics of one frequency. But what if one wants to visualize 50Hz as well (f_{PAL} compared to f_{NTSC} on TVs)? In this case it is necessary to find a harmonic of both 60Hz and 50Hz. As this is easy for two frequencies, it raises problems for multiple frequencies which may even be variable. To solve this problem **schedulers** are introduced.

It separates the task of toggling the lights on and off from the loop itself. Assigning it to external functions and giving the task of calling those functions to the scheduler. Basically it is a forever-loop that checks various clocks and calls the associated function if the timer is due. This can be implemented using classes (which will be shown at the end of this chapter but an exemplary scheduler can be seen in `LED_scheduler.cpp`). If a task is defined as a class that contains a timer as well as the function (or a function pointer) the scheduler can check the state of the timer and only if this returns 'ready' it will call the function of this task.

Using this, a whole world of **scheduling algorithms** becomes available [11].

In the LED example all tasks were given the highest priority implicitly. But this may raise some problems for tasks that take a significant amount of time to be completed - especially if the importance of these tasks varies.

To deal with this, **earliest deadline first scheduling** is introduced: All tasks which are ready to be executed are compared regarding their deadline. For periodic tasks this would be the moment when the task has to be called again. Only the task with the earliest deadline is called first and so on (figure 3.1c).

Contrary to this one can give each task a fixed priority (possibly an attribute of the task class) and the task with the highest priority is started first. This is called **fixed priority scheduling** (figure 3.1a).

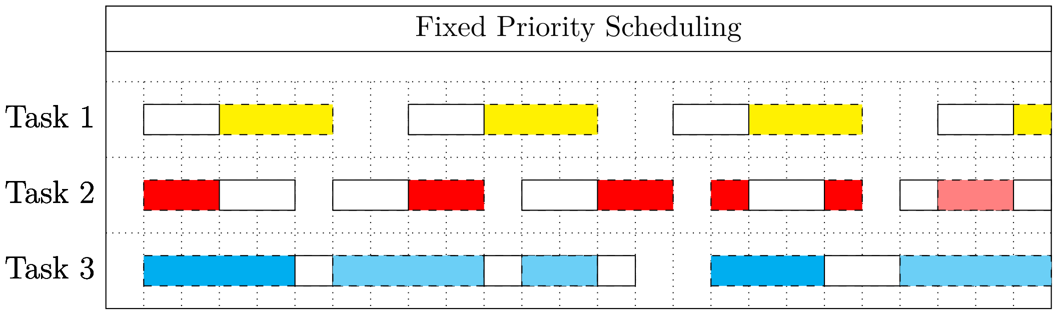
Lastly there is the possibility for **rate monotonic scheduling**. This uses the periodicity of a task as its priority. The shorter the period of a task the higher is its priority (figure 3.1b). Obviously this is closely connected to earliest-deadline-first scheduling. But differences become more significant if we introduce **preemption** to schedulers.

3.3 Preemption

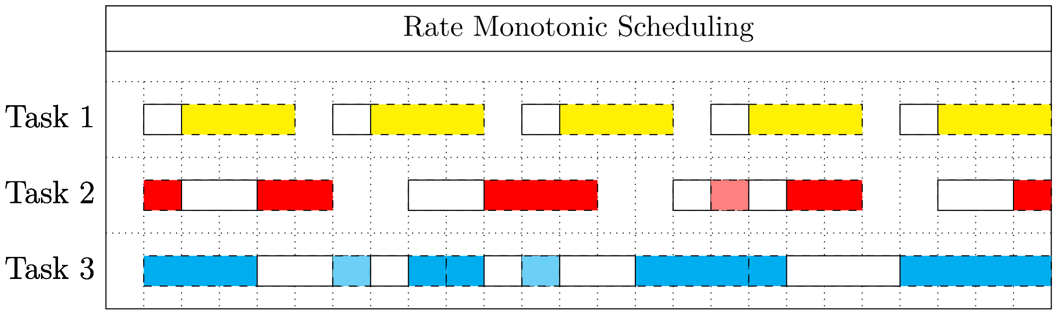
Preemption is the process of suspending or interrupting a running task for a higher priority task. This has to be done by the scheduler and normally the suspended task is resumed the moment the higher priority task has finished.

This is of special importance for larger scale systems where multiple processes might run at the same time. There are even some operating systems which offer preemption in their kernel (appendix 6.1). For small scale system this might not be practical as every interrupt creates **overhead** and slows down the overall system. Preemptive schedulers are most often realized on larger scale systems on a kernel level since multiple tasks or inputs must be handled at the same time. Overall preemption increases responsiveness for high priority tasks at the cost of overhead.

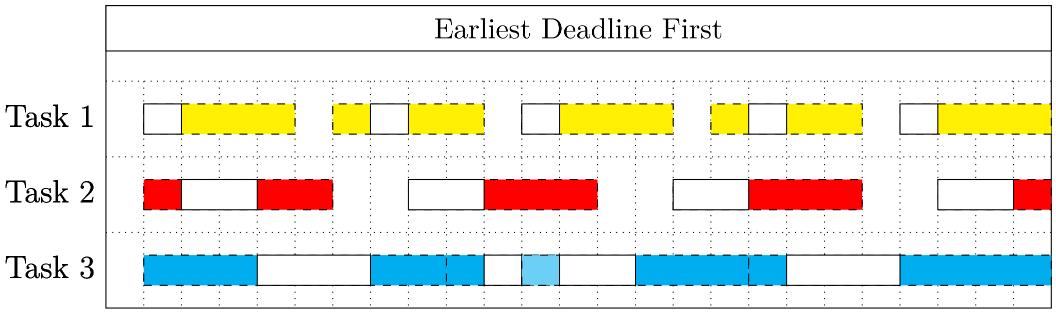
Overhead means the time it takes to switch tasks, which elongates overall execution time and also the resources which are blocked by the suspended task. This includes the stack as well as the function pointer of the suspended task which must remain in the memory. Therefore memory can be clobbered up if too many tasks are suspended.



(a)



(b)



(c)

Figure 3.1: Gantt diagrams for different schedulers. The dashed boxes show starting time and deadline, the white blocks symbol actual computing time. This example includes preemption.

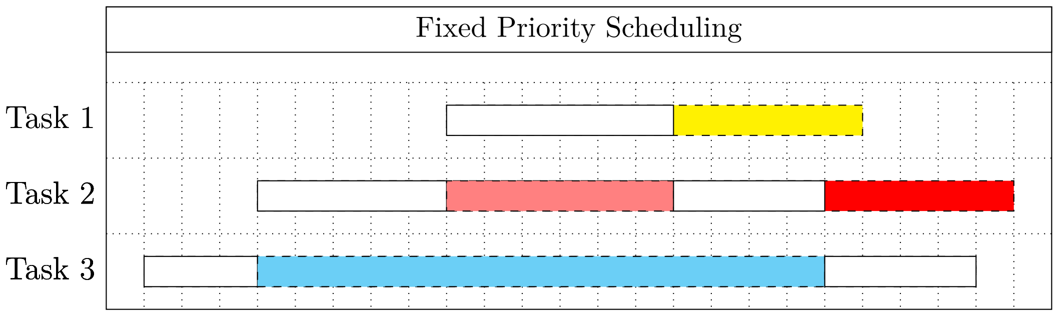


Figure 3.2: Priority Inversion, where Task 1 would rely on a result from Task 3, which is blocked by Task 2.

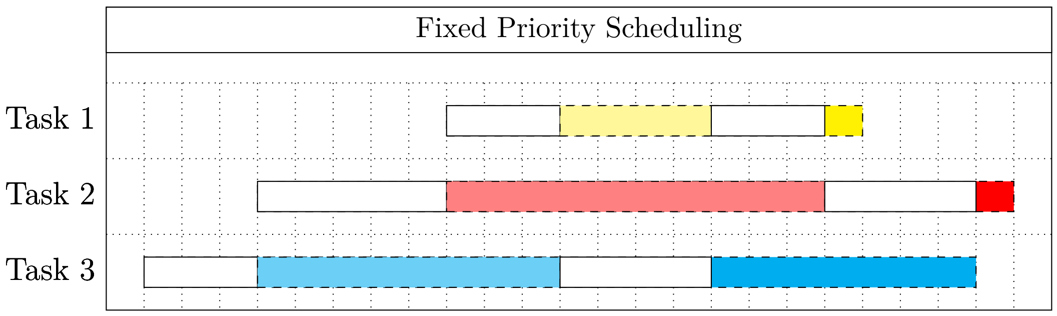


Figure 3.3: Priority inheritance gives Task 1's priority to Task 3 which can then be continued. The result becomes necessary in the middle of Task 1.

3.4 Other Multitasking Issues

Now imagine multiple tasks in a highly sophisticated system, which partly rely on results of one another and are given different priorities. It is easily imaginable how this could render problems if a higher priority task needs the result of a lower priority task which itself is blocked by the higher priority task or any other task in between.

This is called **priority inversion** and is a common problem with real time systems. It actually happened to the NASA in their Pathfinder mission but could be fixed later on [6].

To solve such a problem one introduces **priority inheritance**. This simply allows the higher priority task to give its own priority level to a lower level task which it relies on. Alternatively one can develop a scheduler that uses **cooperative multitasking**. This kind of scheduling lets the tasks communicate between themselves to decide for which task to be started next and avoiding preemption altogether which prevents priority inversion. A nice side effect of this is the increased yield of the program since overhead is reduced. Cooperative multitasking can become very sophisticated especially when trying to find an ideal schedule.

It was hinted at a class which combines all information for a task as well as function pointers. These classes are called **task control blocks (TCBs)**. A basic example of this is `TCB.h`. This is fairly straight forward since important information is grouped as attributes and combined with a special task class (see `task.h`) that holds a function.

All of these techniques and problems are of importance when using multitasking and scheduling. The given TCB example is only one way to implement this and there are various libraries that feature TCBs for real-time programming (appendix 6.2). To complete this thought it is necessary to take a better look at memory.

Chapter 4

Memory Management

First off it is necessary to understand the general structure of memory which consists of a so called **stack** and a **heap**.

The stack is the equivalent of a short-term memory of a system. Data is managed in a last-in-first-out manner. The stack is highly efficient and continuously used in a programs execution.

The heap on the other side is a dynamically allocatable area of the memory. This area is separated into blocks of a certain size. Data is then split up into several of these blocks if necessary or fills up only a fraction of the blocks size if it is smaller. These blocks are also called **memory pages**. This so called **data fragmentation** must be considered when handling memory by hand, which is often the case for embedded systems.

Still, one also needs to keep memory management in mind when developing a real time application of a general purpose system, especially when multitasking is used. Since memory is a limited resource and every task needs some memory there is a mechanism in place that prevents the memory from overflowing. This is called **swapping**. Effectively some data blocks are taken from the memory and temporarily written on the hard drive instead. Because of the massively lower latency and speed of hard drives compared to RAM, this results in increased latency for programs that use this section of the RAM.

4.1 Memory Locking

Now, to prevent this there is the possibility to lock data in its place in memory. **Memory locking** prevents the locked data from being swapped out which forces other data to be swapped out. Using the functions `mlock` and `mlockall`, certain addresses or every address that is used somewhere in the program are locked. Of course this must be handled very cautiously. If one locks all available RAM addresses, this will ultimately result in errors on a GP system or possible system failure on an embedded system. It is therefore recommended to also use the `munlock` and `munlockall` functions respectively.

This can also be applied to the blinking LED example (`LED_wo_mlock.c`. If the program is started followed by a so called 'memory eater' (`eatmem.c`), one can observe how the blinking should stop for some moment and then start again (since the swap is not deterministic -remember this is actually a GP machine- it is at times very hard to observe). This is the point in time where an instruction or other memory reference is requested, which has been swapped out before. If `mlockall()` is added with the flag `MCL_FUTURE` all data pages which this program is going to use are locked (example `LED_mlock.c`). Rerunning the modified example, the blinking should not stop even if the memory eater worked for a while. This solves one of the main problems of memory for real time multitasking (namely **competitive use of memory**) [7].

The problems for **shared use of memory** emerge for non-real time applications as well as for real-time ones through the parallel use of memory by multiple tasks.

If two tasks were to access the same section of memory at the same time they might interfere with one another unless these processes **mutually exclude (mutex)** one another. To achieve this there exist mechanisms such as the 'Peterson Algorithm' that manage access rights for **critical data sections**. These mechanics often use so called **semaphores** to signal to other tasks that a section in memory is currently used. Alternatively one can use existing solutions such as `atomic_store` and `atomic_load` from the atomic library. These functions are well defined for simultaneous requests. Ultimately this is also parallel computing and should not be discussed further in this report.

Nonetheless, mutual exclusion can also introduce latency problems for real time applications. It is therefore often necessary to introduce similar mechanisms as described for schedulers 3.2 to allow prioritized tasks to access relevant memory as fast as possible.

4.2 Cache Invalidation

This goes hand in hand with a feature that is often found on GP systems - the **cache**.

It is meant to decrease latency of a section of data which is accessed repeatedly. By 'caching' data closer to the CPU in highly specialized memory the latency increases while transfer speeds increase.

Although this usually accelerates processes, which might seem good for real-time applications, it interferes with the other mayor principle of real-time computing - determinism.

Because the developer does not have control over the cache and its data, it can not be guaranteed that some data is available in the cache. This is problematic when developing for real-time purposes and is contrary to the principle of determinism. If one is to develop an application while the cache is active, the deadline might be met if certain sections of data are available in the cache. If this is not the case for some reason, the program will fail. It is therefore recommended to deactivate the cache during development and also keep it deactivated since only then determinism can be guaranteed in this section.

The examples did not encounter such problems yet even though the cache has not been invalidated but these problems often only become apparent after long time use. So **cache invalidation** is done on the Raspberry Pi by altering the `boot/config.txt` file by adding an instruction to deactivate the cache on the Raspberry Pi (listing 4.1).

Listing 4.1: Cache invalidation on a Raspberry Pi in `boot/config.txt`

```
enable_l2cache=0
```

Chapter 5

Summary and Outlook

This concludes this report on real-time programming.

It was meant to give as much of an overall - but short - look into different aspects of real-time programming as it was possible. Although it started with many examples it ended with more abstract aspects of real time programming. This hints at the complexity of actual real time development which is an important subject of computer science.

The examples also do not provide much more than the mere idea behind some of these concepts. There are far better implementations that feature more complexity and are tested completely.

It has been said that everything is real time to some degree, and although this is true it should be obvious that it is not always worth the effort. Real-time applications need extensive testing, especially regarding determinism. If one where to develop a real-time application without proper testing they are likely to run into some problems in the long term. Therefore it should be decided whether a task needs to support real time and then how to implement it into a whole system and its environment.

These constraints most often affect sub-tasks or background tasks which need to meet a certain deadline and guarantee determinism. For this as well as for small and embedded systems this report features some ideas which might be helpful. For large systems and general purpose systems on the other hand the range of tools which is available by far exceeds the extent of this report but a mere idea of what is possible should have been provided.

Real time programming will be at least as important in the future as it is today and current development in the digital sector increases the need for real time systems even more. This is obvious when thinking about two of the prime examples of real-time programming which are robotics and automotive driving, or in general automation. Automation gains ever more traction and creates new markets with the need for some key aspects of real time computing which were partly featured in this report.

Chapter 6

Appendix

6.1 Real Time Operating Systems

When developing on GP systems it is worth and often necessary to use so kind of **real time operating system (RTOS)**. Though not the OS itself but the kernel is basically the problem with most common OSes. The Linux kernel for instance does not support preemption which is important for many real time applications.

Although there are some efforts to add real-time support to the Linux kernel which even work on the Raspberry Pi [4], these kernels do not offer full compatibility with many OSes. An example for this is available on github (link) This raises the need for complete OSes that support RT in every aspect. Some of which are QNX, Xenomai and FreeOS. Those and others differ in how they exactly implement features such as scheduling in the kernel. They mostly differ in offering preemption or different schedulers. The kernel is still essential to everything as many systems have a so called 'micro-' or 'nanokernel' which handle only the very basic tasks or offer nothing else but scheduling, although this can be implemented differently as well as in RTLinux. Typical other OS tasks are then implemented as user processes or implemented otherwise outside the kernel. This allows for some operating systems to be compatible to typical embedded systems as well, while others are optimized for machines as large as a mainframe.

Ultimately it is mostly recommend to use one of these operating systems for real time applications if it is not possible to realize with an embedded system.

6.2 Libraries Featuring Real Time Tools

A different approach is real time are libraries that add some real-time capabilities to GP systems or embedded systems. Classes like the TCB are a good example for this. They often offer a multiude of tools such as more exact clocks etc. as well and help developers in different aspects.

The most prominent of these libraries is the **POSIX library**. POSIX stands for Portable Operating System Interface and was initially developed to support applications on different UNIX-based systems. it features many functions and classes that are needed for various applications. Some of these are timers, signals, I/O Port Interface and Control...[5]

This includes tools for real-time as well as for parallel computing. It thus became sort of a stnadard library when developing real time systems even if they are not meant to be ported to different UNIX systems.

6.3 Test Set-Up

The test set-up consists of a Raspberry Pi 3 and a total of three external LEDs as well as three resistors. Quite possibly any combination of an LED and a resistor that works with 3 Volts will work. In the author's set-up generic LEDs from a set and resistors with 330 Ohms were used. Namely the ports 17, 22 and 27 should be used with the provided examples. A good explanation on how to connect the LED can be found online [13].

The testing files can be found on github (link). The repository also features the compiled files. Generally, the `*.c` files should be compiled using `gcc` while the `*.cpp) should be compiled using \lstinline{g++`.

Bibliography

[1] Brown, E., Real-time linux explained, and contrasted with xenomai and rtai, accessed 2017.10.15.

[2] Buttazzo, G. C., *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed., Springer Publishing Company, Incorporated, 2011.

[3] Computer History Museum, Real-time all the time, accessed 2017.10.15, 2017.

[4] Durr, F., Raspberry pi going realtime with rt preempt, accessed 2017.10.15, 2015.

[5] Gallmeister, B. O., *POSIX.4 Programmers Guide - Programming for the Real World*, 1st ed., O'Reilly Associates, Inc., 1995.

[6] Jones, M., What really happened to the software on the mars pathfinder spacecraft?, accessed 2017.10.15, 2013.

[7] Kay, J., Introduction to real-time systems, accessed 2017.10.15, 2017.

[8] Kormanyos, C., *Real-Time C++ - Efficient Object-Oriented and Template Microcontroller Programming.*, I-XXIII, 1-357 pp., Springer, 2013.

[9] Lipasti, M., Lecture 6: Embedded processors - embedded computing systems, accessed 2017.10.15, 2007.

[10] Nilsson, A., Datorteknik och realtidssystem, accessed 2017.10.15, 2017.

[11] Petters, S. M., Real-time systems, accessed 2017.10.15, 2007.

[12] Schnabel, P., Swapping beim raspberry pi einrichten und deaktivieren, accessed 2017.10.15, 2017.

[13] The Pi Hut, Turning on an led with your raspberry pi's gpio pins, accessed 2017.10.15, 2015.