Real Time Systems by Fixed Priority Scheduling[1]

Ken Tindell and Hans Hansson

Department of Computer Systems, Uppsala University

with contributions by Jens Larsson

October 24, 1997

Contents

Chapter 1

Introduction

1.1 Background

The first thing most books in real-time systems do is define "real-time". Definitions don't usually help very much, but to keep the tradition we will give a definition:

> *Real-Time System: Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to the same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.* (Oxford Dictionary of Computing)

It is easy to see why definitions are not very helpful, because with *all* systems, we care about the time the output is made: if you are using a workstation and enter a command, you care that you don't wait more than a few seconds for the response. If you send a piece of e-mail you care if the e-mail takes more than a few hours to arrive. If you are controlling an air-bag in a car you care if the air-bag isn't activated a few milliseconds after a crash. But when we come to engineer these systems, we take very different approaches in meeting the timing requirements.

We can make a distinction between real-time systems, and we usually pick two categories: *broad* and *narrow*: a broad system is where the typical demands on the system can easily be met without any special techniques or analysis. A narrow system is where we have to carefully design the system to meet the demands, because to do otherwise would lead to failure. In this book we are concerned with techniques for solving real-time problems for narrow systems.

We can also classify real-time systems into two types based on how much we care that the timing requirements are met: for *hard* real-time systems, we care very much if the system doesn't respond on-time. For example, with our air-bag example, the driver will be hurt if the air-bag is not inflated in time (most hard real-time systems are also *safety critical*, where life is at risk if timing requirements are not met). The other type of real-time system is a *soft* real-time system. With these systems, we might occasionally fail to meet our timing requirements, but everything is still OK. These systems are more difficult to specify: usually a control engineer will say that a system is hard real-time when in practice it has soft timing requirements. This is because it is often too hard for the engineer to specify which timing requirements can be relaxed, and how they can be relaxed, how often, and so on. This book will mostly focus on hard real-time systems, but will talk about how we can mix hard and soft real-time components in a single system.

Most narrow real-time systems are dedicated to controlling some piece of equipment in the real world. Usually, the piece of equipment won't work without the computer, and the computer is regarded as just a component of the whole system. For this reason, these computer systems are usually called *embedded systems*. Generally when people say *real-time*, they really mean *embedded real-time*. Most hard real-time systems are also embedded systems, and so in this book most of the techniques will be geared towards engineering embedded systems.

The techniques we use to engineer embedded systems must nearly always be *efficient*: we would like to squeeze as much out of the computing resources as possible. This is to keep the costs as low as possible: if the embedded system is going to be mass produced (for example, it might be a washing machine control system) then a small saving on the computer can mean a large saving overall. For example, if we can use a slow processor, saving just $1 over a more expensive processor, then after a million washing machines have been sold we have saved a million dollars. Surprisingly the problem of using slow cheap components also occurs with expensive one-off things like nuclear submarines. This is often because the computers used must be tried-and-tested, and known to work properly, which means using old (and hence slow) technology.

1.2 The Concurrency Problem

Let's turn to the problem of *concurrency* now. The real world is concurrent (parallel, in fact): in any complex system there are lots of events that could occur at the same time. For example, in a computerised car, there could be a sensor monitoring acceleration to detect if there has been a crash (and to activate an air-bag if there has). There could also be a sensor monitoring the brake pedal to see how fast the driver wants to stop. There could also be indicator lights that must be turned on and off regularly. There could be sensors from the wheels indicating the type of road surface so that the braking system knows how to respond. And so on. Each of these devices causes or requires events: the crash sensor causes a "crash detected" event, the indicator lights require regular "on" and "off" events, and so on.

If we have only one processor to handle these events then the processor must execute concurrently: each event requires some amount of computation time, and has some time window in which the processor must respond. So, for example, responding to the "crash detected" event might take 1ms of CPU time, with the computation required to be completed before 10ms and not before 4ms (say). The indicator light "on" event might take a small amount of time to be processed (0.1ms of CPU time, say), but have a long time in which to complete (say 100ms).

Notice how we have *sporadic* events (such as "crash detected"), where we cannot predict when they will occur, and *periodic* events (such as the events to make the indicator light flash on and off). We will see later how a sporadic event can be transformed into a periodic event by *polling*. Notice also how the timing requirements are typically expressed: as a bound on the time taken to perform some computation. We normally refer to this bound as a *deadline*[1].

For the "crash detected" event we have a deadline on how long we can take to respond, but we also must not respond *too early*: we have a window in which a response is required (sometimes this is known as 'just in time' scheduling). If we respond quickly to the "crash detected" event and activate the airbag too soon, then it will have inflated and then deflated by the time the driver hits the steering wheel (airbags must deflate quickly so that the driver can still steer the car if an airbag goes off by accident).

Because we have a number of pieces of computation to choose to execute when a number of events occur, we are left with a problem: which computation to execute, and when? It is a problem because there are clearly good choices and bad choices. Consider the following example:

> We have two events e_1 and e_2, which require computation time 5ms and 10ms, and have deadlines of 20ms and 12ms. Assume both events occur at the same time (time t, say): we must choose which event we will process first. If we process e_1 first, then e_1 will be processed by time $t + 5$. Then we switch to processing e_2, which will be finished by time $t + 5 + 10 = t + 15$. The deadline for event e_2 has been missed (see Figure 1.1). But if we had processed event e_2 first, then all would have been OK: e_2 would have finished by $t + 10$, and e_1 would have finished by $t + 15$, and both deadlines would have been met (as illustrated in Figure 1.1).

You may ask the question "How do we know how much CPU time is needed to respond to an event?" Answering this question is a whole field of real-time engineering in itself. Briefly, we either measure the code on

[1]A 'deadline' gets its name from a piece of rope surrounding a prisoner-of-war camp: all the prisoners were warned that anyone crossing the line would be shot dead

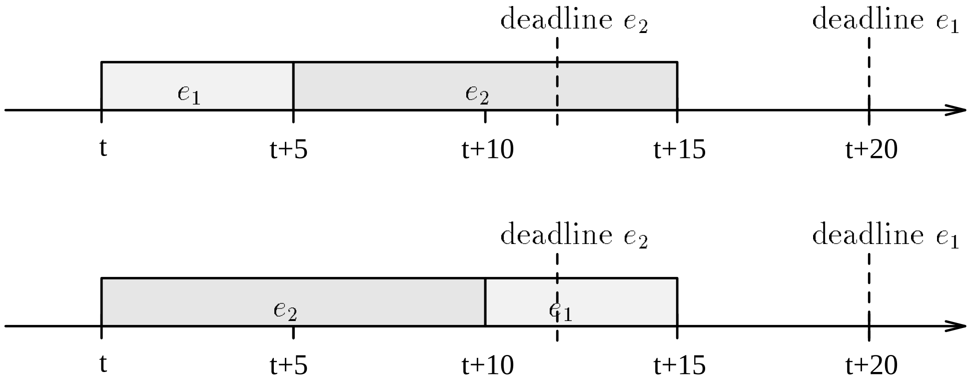

Figure 1.1: e_2 misses its deadline if processed after e_1 (top). Both deadlines are met if e_2 is processed before e_1 (bottom).

a test-rig (with all the problems of testing and coverage), or we attempt to analyse the behaviour of the code, looking at the machine code and any high level language code that it was produced from. We'll describe these two processes in more detail later in this book (in Chapter 2).

We may have other constraints on how we can choose between the processing of events: if we can pre-empt the computation in response to an event, then we must be careful how data is shared between the processing code for different events. For example, we might be in the middle of updating some data-structure (eg. a linked-list) when a switch occurs, leaving the structure in some unknown state. If the code we have switched to tries to access the same structure then there could be an error (eg. the code might fall off the end of a broken linked-list). This is a well-known problem in concurrent programming, and there are several techniques for overcoming it (eg. using semaphores to guard access to the data structure), most are known as *concurrency control* techniques. We'll have a lot more to say about concurrency control later in the book.

The choice about which event to process is generally known as *scheduling*, and most real-time engineering is concerned with scheduling, scheduling algorithms, and analysis of those algorithms.

1.3 The Scheduling Problem

We can break down a scheduling approach into three activities: *off-line configuration*, *run-time dispatching*, and a priori *analysis*. Scheduling approaches mostly differ in how much effort is required in the three activities.

The run-time dispatching activity deals with the actual switching between computations for different events at run-time. Most scheduling approaches assume an underlying *tasking* system on a processor. A *task* is a thread of control, invoked by some event. The usual model is that task executes a piece of code in response to the event (the code can make inputs and outputs, and could cause other events in the system). The task then terminates, ready to be re-awakened when the event reoccurs. So the run-time dispatching algorithm decides when to switch between tasks. To make the decision it can use knowledge about the current state of the system (eg. what the current time is) and information given to it off-line (eg. a set of priorities for tasks). We can switch between tasks in two ways: *pre-emptively* and *non-pre-emptively*. If we switch pre-emptively we are allowed to stop the task we are running now, and switch to another task. Later we switch back to the task we interrupted. With non-pre-emptive switching, once we start running a task we must let it finish before changing doing computation for some other event [2].

The off-line configuration activity generates the static information for the run-time dispatching algorithm to use. In some scheduling approaches, this activity is the heart of the approach: the configuration information consists of a table indicating exactly which task should be run and when. The run-time dispatcher merely has to follow the table, indexed by the current time. In some other scheduling approaches, this activity can be trivially small, or not done at all: the dispatcher uses run-time information, such as task deadline, to choose which task to run at any given time.

Finally, let's talk about the *a priori* analysis. So far, we haven't said

[2] In the strictest sense, 'pre-empt' means to go first ahead of something else, but in computing it generally means 'to interrupt'

much about hard real-time *vs* soft real-time (apart from defining the terms). With hard real-time systems we must schedule the processing of events such that all the deadlines will be met. With soft real-time systems the scheduling algorithm is allowed the flexibility to decide not to meet a deadline. Because this book is focussed on hard real-time scheduling, we will not talk about soft real-time until much later.

With hard real-time scheduling we must always meet our deadlines. But we can have situations where it is impossible for the run-time dispatching algorithm to ensure that all deadlines are met: if, for example, two events occurred at the same time, both requiring computation time equal to their deadlines, then there is no switching pattern that will enable both deadlines to be met. So, what we must do is look at the system *before* it starts running and try to see if a set of circumstances could arise where deadlines would be missed. If we do find that it's possible for deadlines to be missed then we shouldn't start the system running because there's a timing bug. So we have to have *analysis* that will examine the system and tell us if all the timing requirements will be met.

The analysis needs to know what off-line configuration information, and what the behaviour of the run-time dispatching algorithm will be. Of course, it may be very difficult to derive analysis that gives an complete answer to the question "Will all deadlines be met?" The analysis is *sufficient* if, when it answers "Yes", all deadlines will be met. It's pretty easy to get sufficient analysis: we always say "No"! The analysis is *necessary* if, when it answers "No", there really is a situation where deadlines could be missed. We say that the analysis is *exact* if it is both 'necessary' and 'sufficient'.

We require that the analysis is 'sufficient' (if the analysis weren't sufficient then there would be no point in having it in the first place!). But we would also like the analysis to be as close to 'necessary' as possible (the 'necessary' property measures the quality of the test) so that we don't have many systems that are actually schedulable but that the analysis has deemed unschedulable [3].

[3] where 'unschedulable' means 'the scheduler will not find a way to switch between the processing of events such that deadlines are met'

So, what scheduling approaches are there? This book is mostly concerned with one algorithm called *fixed priority pre-emptive scheduling* (or just *fixed priority scheduling*, FP), but there are lots of others, the most popular being: *Static Cyclic Scheduling* (SCS). We'll talk about this approach briefly in Chapter 3, and then we'll look in detail at the fixed priority scheduling approach.

1.4 Testing and Integration

The *testing* of a real-time system is a major problem: finding test data for ordinary non-real-time systems is difficult enough (and concurrent systems are notoriously hard to test), but when we have to test a system by checking all the possible times *when* events happen and data is processed, then it becomes virtually impossible to test all the different behaviours.

We often build embedded systems with teams of people, not just one lone programmer. But we cannot test the timing properties of a component by itself. We can't test that a task meets its deadline until all the other tasks in the system are there too (otherwise we might find that the task met its deadline when there was nothing else to do, but misses its deadline when the other tasks are there).

Because we can't test a real-time system very easily, different people cannot know that their piece of the system is going to work until all the pieces are put together into a final system. All this means that most of the effort in building a real-time system has traditionally gone into testing, bug fixing, re-testing, and so on. Problems with a project don't show up until the end, and by then it may be too late to solve them. So we need to be able to build components in isolation, and know that when we bring them together into the final system, we don't suddenly see hidden timing problems.

The analysis part of the scheduling algorithm helps us here: we can build components separately (eg. different programmers can write their own tasks), and we can test them separately. We can apply the analysis to the scheduling approach that will be used with the tasks, and see if the timing constraints are met. The analysis should show up any timing problems, which can then be solved before we put the whole system together.

1.5 Summary

We have seen some of the basic problems and requirements with embedded real-time systems. Firstly, we saw that we are dealing with *narrow* and *hard* real-time systems in this book. Second, we saw that techniques for engineering these systems must be *efficient* and be able to provide *concurrency*. And finally, we saw that *testing* of embedded systems is difficult. We have talked about concurrency and the need for scheduling. We showed how a scheduling approach could be classified into three activities: configuration, dispatching, and analysis. We have talked about why testing and integration in real-time systems is hard, and how analysis of the scheduling behaviour can help with testing.

Chapter 2

Estimating Program Execution Times

2.1 Background

The most basic question we need to ask, before we worry about anything else to do with real-time, is "how much CPU time does this piece of code need?" In this chapter we're going to try and show how this question can be answered.

A programmer will typically write high-level language code like this:

```
if X > 5 then
    X := X + 1;
else
    X := X * 3;
end if;
```

How long will it take to execute this piece of code? The simplest approach is to measure the execution. Let's take a look at this.

2.2 Measuring Execution Times

To find the longest computation time required by the code, we compile the source code and get the machine code, which we then link with libraries, and load into the processor of the embedded system. Then we hook up a logic analyser, or an oscilloscope, or some similar device that can measure small times. We then run the code with lots of test data and keep all the measurements. We then say the longest time required by the piece of code is the longest time we measured in the tests.

The word “say” in the above paragraph is the important one: just because we observe a lot of execution times doesn't mean that we're guaranteed to have seen the longest time. This is because we have no guarantee that we have looked at the longest path through the code: usually the length of a path depends on the data the program is working with. Take a look at this piece of assembly language code[1]:

```
     ...
     MOVE (_X),D0
     CMP D0,#5
     BGT L1
     MUL D0,#3
     MOVE D0
     JMP L2
L1:  ADD D0,#1
L2:  ...
```

This is typical of the sort of code generated by a compiler (in fact, it's based on the code actually produced from the high level code given earlier). The execution time for instructions can be different: simple instructions like `ADD` can be done in a fixed number of processor clock cycles, but the `MUL` instruction has a variable number of clock cycles, depending (in the above example) on the contents of register `D0`. If you look in processor handbooks you'll typically find tables saying how

[1] *The assembly language used here is sort of 68000ish*

long each instruction takes to run. For the multiply instruction, it's usually something like "16 cycles plus 2 times the number of '1's in the register."

So, the computation time of the small piece of code depends on what's in `D0`, which depends on what the value of `X` is, which could be anything (read from a sensor, perhaps). When we do the tests to find out how long the code takes to run, we can't be sure that the value of `X` had the largest number of '1's (for example). And, of course, the value of `X` that makes this code run for the longest time may make another piece of code run for a small amount of time.

Cacheing and pipelining in processors can also have a big effect on the computation time of a piece of code. For example, if a piece of code is small enough to fit entirely in the cache then the computation time is usually very much smaller. With pipelining, if the instructions don't execute any branches then the pipeline stays filled, and the code executes faster. When we come to do the testing we might be able to disable pipelining and cacheing, but not all processors support this.

The implementation of the memory on a processor card can affect computation time: if slow memory is used then the processor often inserts 'wait states' until the memory responds. If the computer has memories with different speeds, then the execution time of the code can change depending *where* the program is stored! This happens a lot with 'micro-controllers'. These are small processors with built-in memory (usually 1-8Kb) and IO, but often fitted with external memory too (both ROM and RAM). The external memory is slow (ROM is the slowest), and the internal RAM is usually very fast.

The problem of memory access times gets worse for *multi*processor systems (where several CPUs can share a single piece of memory). For example, with a VME bus, several processors can access memory in different cards attached to the bus. If a processor wants to use the bus (to access a memory card attached to the bus, perhaps because a piece of code is stored there) then it can only do so if no other processors are using the bus. If other processors are using the bus then the processor has to wait. So, the computation time of a piece of code on one processor can depend on how code on other processors executes.

You may say "Well, a small overrun in a computation time probably won't matter much, so a simple testing approach is probably OK. We can take the tested time t and then just add 5% for safety." That would be fine, except we may have a huge difference between the times from testing, and the real execution times. Suppose that during testing we never execute a particular part of the code we are testing. For example, we never execute the `else` part of an `if` statement. But during the operation of the system we might often execute the `else` part. If the `else` part takes a lot of time to run, or if the `if` statement were inside a loop, then the total execution time could be many times bigger than the time we found during testing.

So you can see from all this that testing a system for the worst-case execution time of a piece of code will probably give very poor coverage of the real behaviour of the code, and that we cannot be sure we have found the worst-case behaviour. Of course, if we are dealing with a simple piece of code, with a simple processor (no caches or pipelines, no multiprocessor shared memory, etc.) then we *might* be able to get away with testing.

Aside from the accuracy problem, testing is a bit of a pain to do – we know that testing a non-real-time program can take a long time, but testing a real-time program is worse: whenever we make even a small change to the code we would have to run the whole set of tests all over again, to either check that the total time isn't affected, or to get the new time if it is.

What we really want is an automated and analytical approach: we would like to take our source code, give it to some software tool and say "what's the longest execution time of the code?" Such a tool would also be able to tell the programmer which bits of code ran quickly, and which bits were contributing the most to the total execution time: a sort of 'hotspot' analysis. This would be great for a programmer, because without even running code in a test-rig the tool can help say which bits of code should be re-written to speed-up the program.

Not surprisingly, the idea of analysis is an area that has been looked at a lot over the last 10 or 15 years, and some of the results are making their way into practice. Let's take a look at the basic ideas.

2.3 Analysing Execution Times

The first thing we learn when trying to analyse worst-case execution times is that we cannot work from just the high level language code: there are too many different ways a compiler can translate the high level code into machine code (you've probably noticed that there are compilers that produced fast code, and others that produce slower code). So we must work with the machine code produced by the compiler from the high-level language.

We mentioned earlier that the CPU manufacturers usually give tables of instruction times. The tables give the number of processor clock cycles, the number of reads and writes to main memory, and so on. These tables are given to guide the programmer in writing very fast assembly language programs, but we can use these inside a software tool for working out the worst-case execution time of our code.

So, we ought to be able to just follow through the machine code, adding up instruction times as we go, to give the final execution time? Wrong! To see why, look at this assembly language code:

```
MOVE (FD4E5000), D1
ADD D2, D1
JMP D1
```

The address `FD4E5000` is a 'jump table', containing a list of memory locations (the contents of `D2` are used to index this table). This sort of code is typically generated by a `case` statement[2].

If we came across a conditional branch instruction we wouldn't know what to do either. So a simple examination of the machine code cannot generally work out which code will be executed, and so cannot work out the computation time (of course, if the machine code is hand-written without using branching then we might be able to analyse it). Compiler writers are familiar with some of these concepts: they use the term *basic block* to describe a piece of code with a single entry point, and a single

[2] *a* `switch` *statement, if you're a C hacker*

exit point, with no branching in between (they use these basic blocks to work out how to improve the speed of the program by making changes to the allocation of registers to variables, and so on). So, we can work out the execution time of basic blocks by just adding up the instruction times, but we can't work out how to combine these times[3].

If we look at the high level code and the machine code *together* then we can overcome some of these problems: the compiler knows what a particular basic block is for, and so knows more about its behaviour than we could infer by just looking at the machine code. As an example, take another look at the high level language code we had earlier (the code beginning `if X > 5`), and take another look at the assembly language generated from it. It's pretty easy to see what the basic blocks are. There are three of them, the first corresponds to the boolean expression in the `if` statement:

```
        MOVE (_X),D0
        CMP D0,#5
        BGT L1
```

The second is the `else` part of the `if` statement:

```
        MUL D0,#3
        MOVE D0
        JMP L2
```

The third is the main part of the `if` statement:

```
L1:     ADD D0,#1
```

As we mentioned before, most compilers will identify these basic blocks as part of their optimisation strategy.

We can easily add up the execution times of the basic blocks, since they contain 'straight line code'. Let's imagine we had a table in our processor handbook that looked like this:

[3] *take a look at the 'Dragon Book' (Aho, Ullman, Sethi:* Compilers - Principles Techniques and Tools, *Addison Wesley 1986) for more details on basic blocks*

Mnemonic	Number of cycles	Notes
MOVE	8	
CMP	4	
BGT	4	
MUL D0, #3	—	16 + 2 times # '1's
JMP	4	
ADD	4	

The table for a real processor will be much more complex, taking into account different addressing modes, numbers of memory accesses, etc. The table above is just to illustrate the idea.

With this table we can work out the worst-case execution times of the basic blocks we had earlier. The first block, $B1$, has an execution time of $8 + 4 + 4 = 16$. The second block, $B2$, has an execution time of $32 + 8 + 4 = 44$ (we assume that the word-size on our processor is 16 bits, and so there can be at most 16 '1's in register `D0`). Finally, the third block, $B3$, has an execution time of just 4 cycles.

By looking at the high level source code we can see how to combine these basic block times. The `if` statement we had earlier can be re-written:

`if` $B1$ `then` $B3$ `else` $B2$ `end if`

We can see that we will execute basic block $B1$ and then either $B2$ *or* $B3$. So the longest time the `if` statement can take to execute is:

$$time(B1) + max(time(B2), time(B3)) \tag{2.1}$$

where $time(n)$ is the worst-case execution time (in CPU cycles) of basic block n (eg. $time(B2)$ is 44). The worst-case execution time for our `if` statement is 60 CPU cycles. If our CPU runs at 8MHz then each CPU cycle is 125ns. The worst-case execution time of the `if` statement is therefore $60 \times 125ns = 7.5\mu s$.

Of course, now we have this time for the `if` statement, we can use the time just as we did for the basic blocks. So, for example, if we had a nested `if` statement:

```
if X = 0 then
    if X > 5 then
        X := X + 1;
    else
        X := X * 3;
    end if;
else
    X := 1;
end if;
```

The execution time of the statement beginning `if X > 5` is already known from the analysis. We could also work out the execution time for the `X = 0` boolean expression, and the execution time for the `X := 1` statement, just like we did earlier. Then we could combine these times just as we did for the basic blocks to give the worst-case execution time of the outer `if` statement.

Of course, things are never so simple! Some language features are more difficult to generate rules for than the `if` statement we had earlier. Consider a `while` loop like this:

```
while X > 5 loop
    X := X - 1;
end loop;
```

The compiler will probably generate two basic blocks: one for `X > 5`, and one for `X := X - 1`. We can work out the times for the basic blocks, but we have a problem when working out the execution time of the `while` loop: we don't know how many times we will go round the loop. Without knowing more about the value of `X` we simply cannot say. So how is an automated tool to know?

The simple solution is the one most often taken: the programmer must annotate the source code to tell the execution time analyser the maximum number of times the loop executes. In the Ada language, the programmer can use a `pragma`:

```
pragma LOOPBOUND(5);
while X > 5 loop
    X := X - 1;
end loop;
```

The Ada `for` loop is interesting, because the programmer doesn't need to include a loop bound: the compiler is able to work out the bounds. This is because the Ada language rules state that with a `for` loop bounds must be known at compile-time.

In other languages such as C, the programmer has to use comments with a special format that the execution time analyser can read, eg:

```
/* WCET: loopbound=5 */
for(;x > 5;x--)
  body;
```

Now, what happens if the annotated loopound is violated during execution? Clearly, for simple cases this can be detected by static analysis, eg. if the loop-variable initially is assigned a constant value (and not altered in the loop body). In the general case it is not possible to check the validity of annotations. If, for instance, the loop-variable is assigned a value read from a sensor we can only deduce that the variable is within its range (which might be 16-bit integer). A possible solution to this problem is to not only provide a loop-bound annotation, but also specify an emergency action that should be taken if this bound is violated. As an illustration consider the following example:

```
/* WCET: loopbound=5 */
for(;x > 5;x--)
  body;
/* ON_OVERRUN */
  emergency_action;
/* END_OVERRUN */
```

In this case, the loop will be broken after five iterations (regardless of the outcome of the test), and the `emergency_action` executed. The longest execution time for the above code will be

$$5 * (time(\texttt{x > 5}) + time(\texttt{x--}) + time(\texttt{body})) \; + \; time(\texttt{emergency\_action})$$

The proposed solution has the potential of giving more accurate wcet-bounds, but it creates a new pain, namely that of writing an `emergency_action` that solvs the initial problem without completing the `for` loop.

Also, there may be loops in the machine code where there are none in the high level source code. For example, the programmer might write:

```
X := Y;
```

where `X` and `Y` are records. The machine code generated might copy each byte of `Y` to `X`, using a loop. The loop will be executed once for each byte of `Y`. But the compiler knows how big `Y` and `X` are, so knows how many times the loop is executed. This means that the programmer doesn't need to say anything about the generated loop (which is good, because most of the time the programmer wouldn't even know there was a loop in the machine code generated for the assignment statement!).

For the same reason as for `while` loop statements, recursion cannot normally be analysed. But because it's a lot harder for a programmer to work out how deep a recursive call can get, most analyser tools simply ban recursion altogether.

There are usually other restrictions on what an analyser tool can cope with. For example, dynamic memory accesses are usually banned, because the time taken to manage the memory isn't known. For example, when using `new` in Ada, the heap management software must find some spare memory to allocate to the variable. But the time taken to find the memory generally isn't known: this is because the state of the memory isn't known (eg. how much memory is free, what sort of garbage collection needs to be done, etc.). Which means that the time required by `new` function cannot be bounded.

Needless to say, `goto` statements are banned: if the programmer writes these then the dataflow in a piece of code can be very difficult to work out, and so how to combine basic block times is also very difficult to work out.

So what are the problems with the analytical approach, apart from the restrictions on how the programmer can use the language? The main problem is *pessimism*: the time for each instruction is assumed to be the longest (remember when we talked about the `MUL` instruction taking up to 32 cycles?). But in reality, there are usually limits on how the program actually can behave (although we may not be aware of them), and so the actual longest execution time may be a lot less than the theoretical worst-case execution time.

Consider the following nested loop-construct:

```
for(x:=1; x<=10; x++)
  {for(y:=x; y<=10; y++)
      body;
   }
```

The maximum number of iterations in each of the nested loops are 10. Consequently, a resonable execution time analysis tool will consider the worst case number of executions of `body` to be $10 * 10 = 100$. This is clearly a pessimistic estimate! A careful investigation of the nested loops will (due to dependencies) reveal the true number of executions of `body` to be 55.

We discussed a similar problem for the testing approach to finding a bound on execution times: the testing approach can often be optimistic. So either we measure the system, and find a value that could be exceeded, or we analyse the system and find a value that can never be exceeded, but is very much larger than the observed behaviour. We can only really take the analytical approach if we want to be sure that our execution times will not be exceeded.

2.4 Summary

In this chapter we have introduced two techniques for finding a bound on the execution time of a piece of code: *measurement* and *analysis*. There are good and bad points to both approaches, but for critical tasks we can really only use an analytical approach.

2.5 Further Reading

For more information about worst-case execution time analysis tools, you can read the following two reports:

"Using the Worst-Case Execution Analyser", Charles Forsyth, Task 8 vol. D deliverable to European Space Agency 'Hard Real Time Operating System

Kernel Study', ESTEC contract number 9198/90/NL/SF

and:

"Implementation of the Worst-Case Execution Analyser", Charles Forsyth, Task 8 vol. E deliverable to European Space Agency 'Hard Real Time Operating System Kernel Study', ESTEC contract number 9198/90/NL/SF

The first report talks about how to use a worst-case execution time analyser (called *WEA*), which was written to analyse Ada programs compiled for a 68020. The report is interesting because it talks about the limitations of the tool (which language features are not allowed, and so on), and because it shows what a worst-case execution time analysis tool looks like.

The second report talks in detail about how the tool is implemented, and is useful because it indicates how much work is involved writing a worst-case execution time tool.

The two reports are available electronically via `ftp` from the following site:

```
ftp.cs.york.ac.uk
```

in the following directory:

```
/pub/realtime/papers
```

The file names are `task8d.ps.Z` and `task8e.ps.Z` (they are stored as compressed postscript).

Chapter 3

Older Scheduling Approaches

3.1 Introduction

We talked a little in the introductory chapter about what the rôle of scheduling algorithms is, and how we can break a scheduling approach down in to three activities. We mentioned some of the most popular scheduling approaches, and in this chapter we will look in more detail at one of them: Static Cyclic Scheduling.

3.2 Static Cyclic Scheduling

This approach is probably the oldest scheduling approach, used since the 1960's. It is an off-line approach, where the task execution pattern is computed by a configuration algorithm. The execution pattern is stored in a table, often called a *calendar*. At run-time the table is followed by the dispatcher.

You may be thinking "How can all the executions in to the future life of the system be stored? The table would be huge!". What actually happens is that a small table is used (often as little as twenty milliseconds worth of tasking decisions), and then when the table runs out the scheduler goes back to the start of the table. In effect, the table is indexed by the current time modulo the table size. Because the execution pattern repeats cyclically, the approach is known as static cyclic scheduling.

Because the run-time dispatching algorithm is so simple (just looking in to a pre-computed table), the analysis of the timing behaviour is also trivial: we just need to run through the table and check that all the timing requirements are met. The real heart of the approach is the off-line configuration algorithm used to generate the scheduling table. There have been many algorithms developed over the years to generate the table (the most common approach in industry is for the tables to be generated by hand!). Before we talk about some of these algorithms we need to talk about a system model.

We don't need to restrict the way our system behaves just so that we can analyse it – the analysis with the static approach is trivial. Any restrictions come about because of limitations with the algorithm that builds the table, and because of the fundamental limits of static scheduling. Let's look at typical limitations, and talk about the sort of system model that we can use.

Firstly, because we pre-plan when we execute tasks, we cannot permit anything other than strictly periodic tasks: we cannot have tasks being invoked by an external event. The only way we can deal with external events is to poll for them. This is not very efficient, since if we want a short delay between an event occurring and the completion of the processing of the event we must create a task with a short period to poll for it. If the event occurs rarely then the processor utilisation given over to the polling task is much more than the true utilisation required by the event. But we must allocate this time because we never know exact when it will be used. This means that very often we cannot accommodate short-deadline sporadic events simply because we do not have enough processing time to both poll for the event and handle other periodic computation.

As an illustration, consider a sporadic event with an event-handler execution time of 2ms and with a 5ms dedline from even occurence to completion of event-handler execution. If this event occurs at most once every 100ms then its worst case processor utilisation will be 2% ($= \frac{2}{100}$). But since we do not know when the event actually occurs, we must generate a static schedule that polls for the event once every third millisecond. Since we for each such polling must be prepared to execute the event-handler 67% of the CPU-time must be allocated to handle this sporadic event which really only requires 2% of the CPU-time.

A second fundamental limitation with the static scheduling approach is that the table can't be too big (because it takes memory to store the table, and in most embedded systems we don't have a lot of memory spare). The table encodes the shortest repeating cyclic execution of the tasks, so that by going back to the start of the table we don't introduce any 'hiccups' in the system. But with periodic tasks, it turns out that the shortest repeating cycle is the *least common multiple* (LCM) of the task periods. So, if we have tasks with periods 5, 10, and 20 milliseconds, the repeating cycle is 20 milliseconds long. But if we have tasks with periods 7, 13, and 23 milliseconds then the LCM, and hence the table length, is $7 \times 13 \times 23 = 2093$ milliseconds – very much bigger! So what is generally done is that the *required* periods are manipulated to be neat multiples of each other: the periods are made shorter than they need be so that the scheduling table is small. Of course, this means that more processing time than is really needed is used (for example, if we had a task with an execution time of 3 milliseconds and we reduced its period from 13 milliseconds to 10 milliseconds, then the processor utilisation used by the task increases from 23% to 30%).

The algorithm to build the table is limited in how well it can perform. In general, building a schedule such that all deadlines are met is an *NP-Hard* problem[1], and so we can't expect an algorithm to always find a scheduling solution if one exists. However, this doesn't worry us too much, since there are a number of algorithms which can give really quite good results (ie. they can mostly find a solution if one exists).

A simple model, eg. assuming strictly periodic and independent tasks with deadline equal to period, can be easily handled by the static scheduling approach. But the real benefit of static scheduling is that we can have a more complex model. For example, we can allow tasks to share data, and then control the way in which we are allowed to switch between tasks. So, we might have two tasks A and B who both access some shared data structure (such as a linked list). The algorithm that builds the scheduling table must make sure that we never switch to B unless A has completed all its execution, and *vice versa* (these restrictions on how we can switch between tasks are often called *exclusion constraints*). We can generalise this exclusion constraint to more than just a pair of tasks, and allow groups of tasks to share data. The algorithm must not switch from one member of the group to another until the current task has finished executing.

[1] Don't worry about this term if you're not familiar with it: what it means is that it's impossible for the algorithm to always find the *best* result in a sensible amount of time for realistic sized problems

There are some drawbacks to this way of controlling concurrency. We would prefer to control concurrency only when we are actually accessing shared data, rather than over the whole execution of the task. For example, if we had a task A with a large execution time, but that only accessed shared data for a small amount of time, all other tasks in the group cannot be switched to at any time that A is running. This is because without detailed analysis of the code of A, we do not know *when* during its execution task A will access the shared data. Because we are working out a schedule off-line, we have to assume that it could be anywhere, and so cannot switch from task A to another task in the group until A has finished. This restricts the number of potential solutions that would otherwise satisfy all our requirements, and may mean that our off-line schedule builder cannot find any solution at all, and has to give up. What we would prefer is that we use something like a semaphore, where we tell the scheduler when other tasks can access the data, and when they cannot. But we can't do this because the scheduler must follow the fixed execution pattern laid out in the table. We'll come back to this problem later when we talk about fixed priority scheduling. In some systems, we are not allowed to switch away from a task until it has completed (if you remember, we called this *non-preemptive scheduling*). If we make all tasks members of the same group then we get this behaviour.

Another common constraint is *precedence*: we might require that a task X finishes before another task Y is allowed to run[2]. Precedence constraints are very useful for many types of real-time system: typically we want to implement a 'pipeline', where input data is read in to one task, which passes computed results down a pipeline of other tasks, with the last task in the chain sending the results to output ports.

Aside from the most popular technique for building schedules to meet these constraints – human intervention – a popular piece of work is by Xu and Parnas[3]. They use a two-phased approach: an algorithm which at every scheduling point schedules the task with shortest time remaining to its dedline (Earliest Deadline First) is used off-line to build an initial schedule, which is then modified according to heuristics[4] to modify it to try and meet the exclusion and precedence constraints. Other approaches use standard optimisation algorithms (like *simulated annealing*) to take a schedule and

[2] This is a stronger form of an exclusion constraint

[3] In fact, we're using this work right here in Uppsala for a tool that will build static schedules for sending messages on a shared bus

[4] or 'rules of thumb'

'mutate' it according to simple rules until a schedulable system is found[5].

3.3 Summary

We looked at static cyclic scheduling (an off-line approach). We saw that quite flexible models could be allowed, but that the approach is not always efficient.

[5]Of course, the algorithm may not find a solution, and will give up instead

Chapter 4

Fixed Priority Scheduling Basics

4.1 Introduction

In this chapter we will be looking at the *fixed priority* scheduling approach, and look at how we can use it (we'll look at a case study to show how deadlines will be met). Most of this chapter will be devoted to *analysis* of a given set of tasks with fixed priorities (remember we said before how analysis was an important part of any scheduling approach: we must know before the system runs if deadlines will be met).

4.2 Rate Monotonic Scheduling

In the early 1970's an influential paper looked at and produced analysis for *fixed priority* scheduling[1]. This is the paper:

> C. L. Liu and J. W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, *JACM*, Volume 20, Number 1, pages 46 to 61, 1973

[1]The paper also locked at Earliest Deadline Scheduling, which is a dynamic scheduling method that we will discuss further in Chapter 13

In fixed priority scheduling, the dispatcher will make sure that at any time, the highest priority runnable task is actually running. So, if we have a task with a low priority running, and a high priority task arrives (ie. there was some event that occurred, and the dispatcher needs to kick off a task to deal with it), the low priority task will be suspended, and the high priority task will start running (ie. the low priority task has been *preempted*). If while the high priority task is running, a medium priority task arrives, the dispatcher will leave it unprocessed, and the high priority task will carry on running, and at some later time finish its computation. Then the medium priority task starts executing, to finish at some later time. Only when both tasks have completed can the low priority task resume its execution (the low priority task can then carry on executing until either more higher priority tasks arrive, or it has finished its work).

Analysis requires a *model* that only considers some (relevant) limited aspects of the system behaviour. The work of Liu and Layland mentioned above used the following model for fixed priority scheduling: we assume that all tasks have periods, with deadlines equal to these periods; tasks are not allowed to be blocked, or to suspend themselves. Also, we need an extra assumption: we need to say how the priorities are chosen. In their paper, Liu and Layland coined the term *rate monotonic* for how priorities were chosen. What they meant by this term is that priorities should be set monotonically with rate (ie. period): a task with a shorter period should be assigned a higher priority. They also assumed that priorities are unique. Where there are 'ties' (ie. tasks with the same period) they can be broken arbitrarily.

Liu and Layland also made a couple of assumptions that don't actually need to be made. They assumed that tasks are strictly periodic, when in fact the analysis also works if a task i that arrives *at most* once every T_i. They also assumed that task execution times were constant, but in fact they just need to be bounded: the analysis works if a task i executes for *less* than C_i.

In terms of our description of a scheduling approach, we can see that the configuration activity is how we choose the priorities: by the rate monotonic policy. The run-time dispatching activity is fixed priority pre-emptive scheduling (as we've just described). What we need now for a complete approach is *analysis*. Liu and Layland gave some analysis for this approach, with the model we have just described. They gave an equation for a *utilisation bound*: if the task set has a utilisation below this bound then all deadlines will always be met (ie. it is *schedulable*). But the analysis they gave is not *exact*: if the utilisation is higher than the bound, the analysis has nothing to say about whether the task set is schedulable or not. It might be. It might not be. The analysis doesn't say (exact analysis for this model does exist though: later on in this chapter we'll give exact analysis). Below is the scheduling test provided by Liu and Layland:

$$\sum_{i=1}^{i=n} \left(\frac{C_i}{T_i}\right) \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{4.1}$$

where n is the number of tasks in the system, C_i is the worst-case execution time of task i, and T_i is the period of task i. The left side of the equation is the utilisation of the task set, and the right side of the equation is the utilisation bound.

The utilisation bound is generally a lot lower than 100% – for one task the bound is 100%, and for two it is about 83% (you might like to work it out the bound for three tasks). You can see that as $n \to \infty$, the utilisation bound approaches $\ln 2 \approx 0.693$, which is just over 69%. This is the most commonly known result of the work: "if your CPU utilisation is less than 69%, then all deadlines are met." But remember that this is only for the limited *model* that Liu and Layland defined: only periodic tasks, no blocking, and deadlines must be equal to periods.

Since 1973 a lot of work has been done on "rate monotonic", and so now the model that can be analysed is a lot more flexible than was originally defined. If you hear people talking about "Rate Monotonic Analysis" (RMA), or "Rate Monotonic Scheduling" (RMS) they generally mean the more flexible models and better analysis. In fact, these models no longer assume that priorities are assigned monotonically with rate, and so the term is now a misnomer.

We mentioned earlier that the utilisation bound given by Liu and Layland represented analysis that was not exact: some task sets could be rejected that are in reality schedulable. Here's a task set worth considering (we'll come back to it later in this chapter):

Task Name	T	D	C	*RM Prio*
A	52	52	12	Low
B	40	40	10	Medium
C	30	30	10	High

The utilisation of this task set is 81.41%. For three tasks, the utilisation bound calculated by the Liu and Layland equation is 77.98%. So the scheduling test would say "No, this task set might miss deadlines". But in reality, it *is* schedulable as illustrated by Figure 4.1. In fact, this figure presents an exact schedulability test suggested by Liu and Layland: if all tasks meet their deadlines at their "most difficult" invokation then they will always meet their deadlines. Liu and Layland showed that the "most difficult" invokation is when all tasks are released simultaneously. Later in this chapter we will show how this test can be formalised into a set of equations. Equations that can be conveniently implemented in a schedulability analysis tool. Suprisingly enough, the formalisation of Liu and Layland's test did not appear until 1986, thirteen years after Liu and Layland published their paper.

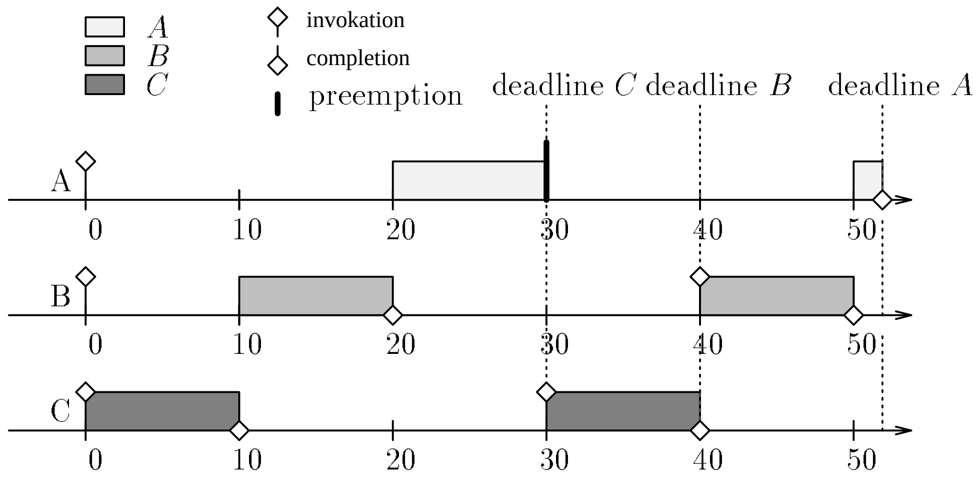

Figure 4.1: An exact test for RMS.

4.3 Exact Analysis

In 1986 Mathai Joseph and Paritosh Pandya wrote a paper which gave exact analysis for the fixed priority scheduling model that Liu and Layland described:

M. Joseph and P. Pandya, *Finding Response Times in a Real-*

Time System, The Computer Journal, Volume 29, Number 5, pages 390-395, 1986

This was an important piece of work because it did more than just provide an exact piece of analysis. It also came up with the idea of calculating the worst-case response times of a task: this is the longest time taken to complete the worst-case execution time of the task. We can extend our notation to include this, and say that R_i is the worst-case response time of a given task i. The table below summarises our notation so far:

Notation	Description
C_i	Worst-case computation time of task i
T_i	Period of task i
D_i	Deadline of task i
R_i	Worst-case response time of task i

Once we have a worst-case response time for each task, the scheduling test is trivial: we just look to see that $R \leq D$ for each task. This means that we can lift the restriction that deadlines are equal to periods (after all, if we know the worst-case response time then we can compare it to any arbitrary value to see if it is small enough for our needs).

A side-effect of being able to calculate worst-case response times is that we can handle *sporadic* tasks very neatly. Sporadic tasks are those that are run in response to some irregular event, but generally have short deadlines compared to their minimum inter-arrival time. Before, with the model that Liu and Layland had for fixed priority scheduling, an infrequent sporadic would either have to be given a long deadline (which we usually don't want to do), or it would have to be given a short period (which would be very inefficient: the analysis would probably then reject most task sets, where in reality they would be schedulable).

Let's have a look at the analysis that Joseph and Pandya produced:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{4.2}$$

Where $hp(i)$ is the set of tasks of *higher priority* than task i. So the summation is over all the tasks of higher priority than i. The function $\lceil x \rceil$ is called the *ceiling function*, and gives the smallest integer $\geq x$. So $\lceil 5.1 \rceil = 6$, $\lceil 5.9 \rceil = 6$, and $\lceil 5 \rceil = 5$.

You might be wondering how to solve the equation Joseph and Pandya produced: after all, the term R_i is on both the left and right hand sides, and cannot be re-arranged to give R_i. It turns out that the equation can be solved by forming a *recurrence* relation, where an iteration is used to find values of R_i that fit the equation. In general, there may be several values of R_i that fit, but the smallest one is the worst-case response time we want. The equation below shows how the recurrence relation is formed:

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{4.3}$$

The term R_i^{n+1} is the $n+1$th approximation to the worst-case response time, R_i, and is found in terms of the nth approximation. We can easily see that approximations will get bigger and bigger, until the iteration *converges* (ie. $R_i^{n+1} = R_i^n$), or exceeds some threshold (eg. D_i). In fact, until it converges R_i will in each iteration increase with at least one of the C_js (since otherwise $R_i^{n+1} = R_i^n$, which would mean thatwe are done). We can start the iteration with any value that is less than or equal to the final value of R_i; we might as well choose $R_i^0 = 0$[2].

With this analysis, a value of R_i is valid only if $R_i \leq T_i$: a response time that is greater than the period of the task is not valid by this analysis. This is because if the response time goes over the end of the period then task i could be delayed by a previous invokation of itself; the analysis takes no account of this, and so is not valid for $R_i > T_i$. This means that the deadline of task i should not be greater than T_i (ie we must have $D_i \leq T_i$). We will later, in Section 11.3, show how the analysis can be extended to allow $D > T$.

Notice how the analysis is dependent only on the task attributes C and T, and the priority ordering of the task set. This means we can replace the simple rate monotonic analysis with this new analysis. Let's apply it to our simple task set we looked at earlier. Here's the task set again:

[2]We could just as well choose C_i

Task Name	T	D	C
A	52	52	12
B	40	40	10
C	30	30	10

First of all, we must work out the priority ordering of the task set. Since it was a task set for the rate monotonic approach, we use that ordering: task C has the highest priority, then task B, and finally task A[3].

Let's find R_A first. The set $hp(A)$ contains B and C. We choose an initial value to start the iteration of zero, so we have $R_A^0 = 0$. The first approximation is then:

$$R_A^1 = C_A + \left\lceil \frac{R_A^0}{T_B} \right\rceil C_B + \left\lceil \frac{R_A^0}{T_C} \right\rceil C_C = C_A = 12$$

and the next approximation:

$$R_A^2 = C_A + \left\lceil \frac{12}{T_B} \right\rceil C_B + \left\lceil \frac{12}{T_C} \right\rceil C_C = C_A + C_B + C_C = 32$$

and the next:

$$R_A^3 = C_A + \left\lceil \frac{32}{T_B} \right\rceil C_B + \left\lceil \frac{32}{T_C} \right\rceil C_C = C_A + C_B + 2 \times C_C = 42$$

and the next:

$$R_A^4 = C_A + \left\lceil \frac{42}{T_B} \right\rceil C_B + \left\lceil \frac{42}{T_C} \right\rceil C_C = C_A + 2 \times C_B + 2 \times C_C = 52$$

and the next:

$$R_A^5 = C_A + \left\lceil \frac{52}{T_B} \right\rceil C_B + \left\lceil \frac{52}{T_C} \right\rceil C_C = C_A + 2 \times C_B + 2 \times C_C = 52$$

And we've converged, because $R_A^5 = R_A^4$. So $R_A = 52$.

[3]Remember: "rate monotonic" means assigning priorities according to period: short period $\rightarrow$ high priority

Let's now work out R_B. We have:

$$R_B^1 = C_B + \left\lceil \frac{0}{T_C} \right\rceil C_C = 10$$

$$R_B^2 = C_B + \left\lceil \frac{10}{T_C} \right\rceil C_C = 20$$

$$R_B^3 = C_B + \left\lceil \frac{20}{T_C} \right\rceil C_C = 20$$

So $R_B = 20$. Finally, let's work out R_C:

The set $hp(C)$ is empty, and so $R_C = C_C = 10$ (that was easy!).

Here's the table for the task set again, only this time with the worst-case response times filled in:

Task Name	T	D	C	R
A	52	52	12	52
B	40	40	10	20
C	30	30	10	10

We can see from the table that the task set is schedulable: $R \leq D$ for all the tasks. This illustrates nicely how the simple rate monotonic analysis of Liu and Layland (the $n(2^{\frac{1}{n}} - 1)$ utilisation bound) is not exact: the above task set is schedulable (as the analysis we've just applied shows), but the task set was rejected by the Liu and Layland test.

The approach so far fits neatly into our way of looking at scheduling approaches (our three activities: configuration, analysis, and dispatching). The configuration activity is choosing the priorities of tasks: the rate monotonic policy (short $T \rightarrow$ high priority). We've seen two examples of the analysis activity: the simple utilisation bound test of Liu and Layland, and the exact analysis of Joseph and Pandya. Finally, the dispatching activity is straightforward: the highest priority runnable task is the one that runs.

We have also seen that the analysis involves a lot of routine calculations even for our very simple example. For more realistic task sets, manually solving the schedulability equations is a real pain. For this reason, and to avoid the risk of making a mistake in the lengthy calculations, we will introduce a simple tool for solving the type of equations we have just seen.

COMMENT: To be provided: a presentation of the tool

4.4 Deadline Monotonic *vs* Rate Monotonic

With the rate monotonic model of Liu and Layland, we had $D = T$ for all tasks. But we said earlier how being able to compute R_i for a task i lets us have arbitrary deadlines. We also said that the analysis earlier was valid for $R_i \leq T_i$ (and hence we must have $D \leq T$), and merely requires a priority ordering to be set. As we might expect, the rate monotonic priority ordering policy isn't very good when we have $D \leq T$: an infrequent but urgent task would still be given a low priority (because T is large) and hence probably miss its deadline. It turns out that another priority ordering policy is better: the *deadline monotonic* ordering policy. Just as its name suggests, a task with a short deadline (D) has a high priority. The deadline monotonic ordering has been proved optimal for systems where $D \leq T$. By "optimal" we mean "if the system is unschedulable with the deadline monotonic ordering then it is unschedulable with *all* other orderings".

Apart from the problem of handling sporadic tasks with tight deadlines, there are other reasons for wanting $D \leq T$: in many systems we want to control *input/output jitter*. What we mean by this is that we sample an input at a regular rate, but take the sample within a tight time window. In many control systems this is vital: if we fail to do this then the control system theory may be invalid, and the controlled equipment may behave badly[4]. Now that we have an approach where $D \leq T$ we can make sure that input and output tasks are assigned deadlines equal to the required time window: this means that if the task set is schedulable then the I/O jitter will be within acceptable tolerances.

[4]If the controlled equipment is something like a jet aircraft then this 'behaving badly' may mean loss of control of the plane, and perhaps ending up with a crash.

4.5 Sharing Resources: The Blocking Problem

One of the restrictions we had with our model was that tasks were not allowed to block or voluntarily suspend themselves. This means that it is very difficult to share resources (such as data) between tasks: recall from our discussion of static cyclic scheduling where we looked at how we had to limit the switching between tasks that share data. We called this the *concurrent update problem*, and said how this leads to *exclusion constraints* in the static scheduling problem.

The concurrent update problem is well known to programmers of concurrent systems, and a number of solutions have been proposed. The most common of these is the *semaphore* (invented in the 1960's by Edsgar Djikstra): a task can only access shared data after it has locked a semaphore (a task can only lock a given semaphore if the semaphore is currently unlocked). Once locked, the task executes code accessing the resource; this code is called a *critical section*. When the task has finished with the resource, it unlocks the semaphore.

If a task tries to lock a semaphore that is already locked, it blocks (ie. stops running) until the semaphore is unlocked by the task currently holding the semaphore. Here is the heart of the real-time problem: a task i might want to lock a semaphore, but the semaphore might be held by a lower priority task, and so task i must be blocked for a time. The analysis of the previous section takes no account of the execution time of tasks with lower priority than i, so what time should we allow? How long will it take, in the worst-case, for the lower priority task to unlock the semaphore?

To answer this question, let's look at an example (all the times in the table are in milliseconds):

Task Name	T	D	C
A	50	10	5
B	500	500	250
C	3000	3000	1000

The utilisation of this task set is just over 93%, and is schedulable *if there is no blocking involved* (the worst-case response times are: $R_A = 5$, $R_B = 280$, $R_C = 2500$).

But let us say that tasks A and C share some data, and each require access to the data for at most 1ms of computation time. A semaphore s guards access to the shared data.

Let's also say that task C is running, and locks s at some time t. Just after task C locks s, task A is activated and (because it has a higher priority), starts running. A little later (time $t+2$) task B is invoked, but because it's not the highest priority task, it doesn't start running. At time $t+3$, task A needs to access the data that's shared between A and C, and tries to lock s. But it can't lock s, because task C is using it. So task A blocks, waiting for s to become unlocked. Task B is then the highest priority runnable task, and starts running. It requires 250ms of CPU time, and so finishes at time $t+253$. Then task C is the highest priority task, and starts running. Just 1ms later (at time $t+254$), task C unlocks s, and task A starts running again. But task A has already missed its deadline! The scenario is illustrated in Figure 4.2.

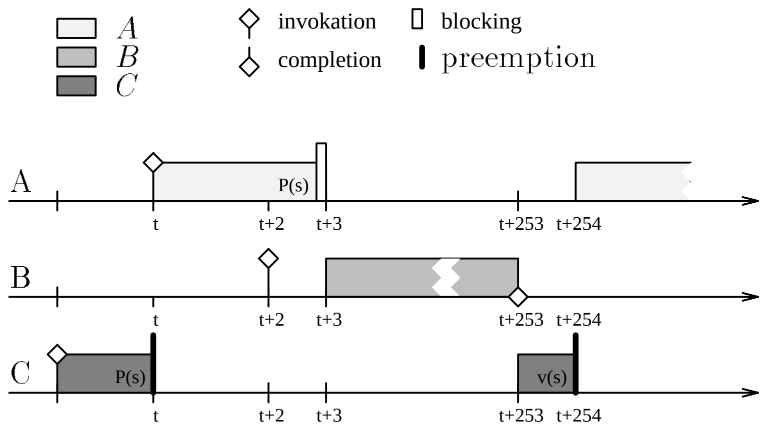

Figure 4.2: Priority inversion.

You should be able to see the problem: in effect, the execution of task B delays the execution of task A, even though it has a lower priority. This effect is known as *priority inversion*, where a lower priority task can delay the execution of a higher priority task. In the example above, the priority inversion is limited to the execution time of task B plus the time that task C can execute while holding s. But in general, it could be worse: there could be lots of tasks with priorities between those of tasks A and C, and task A could be delayed by all these tasks. This could be a very long time. With this blocking problem, the fixed priority scheduling algorithm degenerates to FIFO ('First In, First Out') queuing in the worst-case, which takes no account of urgency.

So it would seem that semaphores wreak havoc with the worst-case timing performance of fixed priority scheduling. Fortunately there is a solution to the problem: *priority inheritance*. It's fairly obvious that for the time that task A wants to lock s but cannot, executing the code in task C is urgent: the execution of task C is contributing directly to the delay of task A. So, why not assign (temporarily) the priority of A to C, so that task C is hurried along? Let's see what happens in our example:

At time t task C locks s, and just after this time, task A is activated and starts running. At time $t+2$ task B is invoked, but is not the highest priority task, and doesn't start running. At time $t+3$, task A tries to lock s, but fails because task C has already locked it. So task A is blocked, and task C inherits the priority of A. Because task C is now the highest priority task, it starts running. At time $t+4$ it has executed for 1ms and unlocks s. Task C now returns to its normal priority, and task A locks s and starts running. Task A completes its execution at time $t+9$, meeting its deadline. The new scenario is illustrated in Figure 4.3

We can see how priority inheritance works: when a task holds a semaphore, it *inherits* the priority of the highest priority task blocked waiting for the semaphore. Notice that the priority of a task is now dynamic, so the name 'Fixed Priority Scheduling' isn't really true anymore. But we still call it this to distinguish it from much more dynamic algorithms like EDF.

But there is still a problem when we come to try and analyse the behaviour of this: we might have deadlock – the equivalent of infinite worst-case response times. So without more information about the behaviour of the tasks, the analysis must report infinite response times for tasks. To illustrate this, consider our example again, only this time with two semaphores, s_1 and s_2.

At time t, task C locks semaphore s_1. Just after this, task A runs and locks semaphore s_2, and then semaphore s_1. It can't lock s_1 because task C holds it, and so blocks. Task C then inherits the priority of task A and starts running. Task C tries to lock s_2, but task A already holds it, and must block. Neither task can execute, and hence we have deadlock.

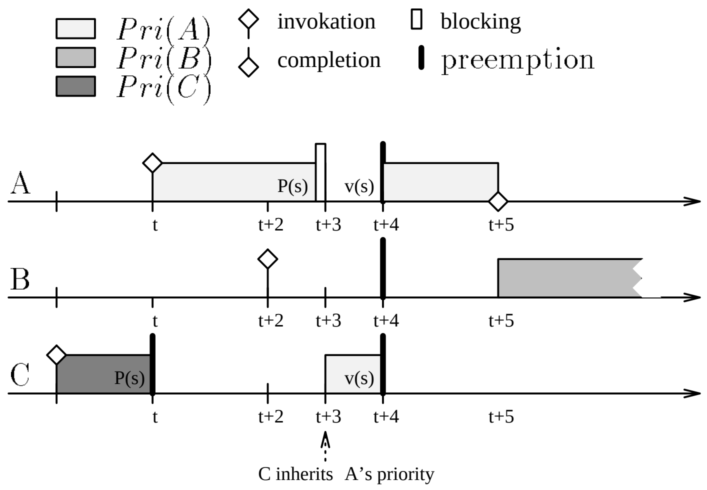

Figure 4.3: Priority inheritance.

It turns out that there is a solution to our problem: a semaphore locking protocol that doesn't suffer from unbounded priority inversion or deadlock, and can be analysed to give the worst-case blocking time experienced by a task. Let's have a look at it.

4.6 The Priority Ceiling Protocol

The protocol is called the *priority ceiling protocol.* The protocol was discovered by Ragnuthan Rajkumar, Liu Sha, and John Lehoczky in 1987. This paper describes it in detail:

> L. Sha, R. Rajkumar, J. Lehoczky, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, IEEE Transactions on Computers, Volume 39, Number 9, pages 1175-1185, 1990

The protocol places a few restrictions on how we can lock and unlock semaphores. The first restriction we have is that a task is not allowed to hold a semaphore between invokations – when a task finishes executing, it must have released all semaphores. This is an obvious restriction: if the task held a semaphore between invokations, a higher priority task would have to wait until the task next executed, which could be a very long time.

The second restriction is that tasks must lock semaphores in a 'pyramid' fashion. For example, if we have two semaphores s_1 and s_2, we can lock them like this:

$$lock(s_1) \ldots lock(s_2) \ldots unlock(s_2) \ldots unlock(s_1)$$

But not like this:

$$lock(s_1) \ldots lock(s_2) \ldots unlock(s_1) \ldots unlock(s_2)$$

This 'pyramid' locking is important because we need it when coming to work out blocking times for tasks.

The protocol assumes that the computation time a given task i needs while holding a given semaphore s is bounded. We'll use the notation $cs_{i,s}$ for this: "the length of the critical section for task i holding semaphore s". It also assumes that a given task i can lock semaphores from a fixed set of semaphores known *a priori*. We shall use the notation $uses(i)$ to be the set of semaphores that could be used by task i.

The protocol uses the idea of a semaphore *ceiling* (hence the name of the protocol!): the ceiling is the priority of the highest priority task that uses the semaphore. We'll use the notation $ceil(s)$ to indicate the ceiling of a given semaphore s, and the notation $pri(i)$ for the priority of a given task i.

At run-time, the protocol operates like this: if a task i wants to lock a semaphore s, it can only do so if the priority of task i is strictly higher than the ceilings of all semaphores currently locked by other tasks. If this condition is not met then task i is blocked. We say that task i is *blocked on* s.

Of course, the priority ceiling of s is at least the priority of i (because s is used by i). If s were already locked then i would not be allowed to lock s because its priority would not be higher than the ceiling of s.

When task i blocks on s, the task currently holding s *inherits* the priority of task i (as we described earlier).

There are some interesting properties of this protocol: firstly, the protocol is *deadlock free*. Secondly, a given task i is delayed *at most once* by a lower priority task. Thirdly, this delay is a function of the length of a critical section (remember, a critical section is the code between a semaphore lock and unlock; the 'length' of a critical section is the time taken to execute this code)[5]. It's not worth going in to the formal proofs of these properties, but it is important to show how the analysis is updated.

We define the term *blocking factor* to be the longest time a task can be delayed by the execution of lower priority tasks, and use the notation B_i for the blocking factor of task i. The blocking factor is easily added in to the analysis we have so far:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{4.4}$$

How do we calculate the blocking factor? A task i can be delayed when it tries to access a semaphore which is locked. This is called *normal blocking*. It can also be delayed when it tries to lock a semaphore but where the ceilings of currently locked semaphores are too high. This is called *ceiling blocking*. It can also be delayed if it is executing ordinary code when a lower priority task inherits a higher priority and executes. This is called *push-through blocking*. So, even if a task does not try to lock semaphores it can still be delayed by lower priority tasks (for example, look at task B in the example of priority inheritance we considered earlier: it is delayed by the execution of task C).

A given task i is blocked (or delayed) by at most one critical section of any lower priority task locking a semaphore with priority ceiling greater than or equal to the priority of task i. We can explain that mathematically using the notation:

$$B_i = \max_{\{k,s \;\mid\; k \in lp(i) \,\wedge\, s \in used\_by(k) \,\wedge\, ceil(s) \geq pri(i)\}} cs_{k,s} \tag{4.5}$$

What this means is: we look at all the tasks of lower priority than task i, and then look at all the semaphores that they can lock, and then select from those only the semaphores where the ceiling of the semaphore has a priority higher than or the same as the priority of task i, and then look at the

[5]We can find this execution time using the methods we talked about earlier in the book

computation time that we hold each of these semaphores for. The longest of these computation times is the blocking time, B_i.

An interesting thing we notice here is that the *lowest* priority task in the system has a blocking factor of *zero*: there aren't any lower priority tasks to delay it!

Let's look at an example of how the priority ceiling protocol works. Assume we have three tasks, A, B, and C.

Task Name	T	D	C	priority
A	50	10	5	High(H)
B	500	500	250	Medium(M)
C	3000	3000	1000	Low(L)

We also have three semaphores, s_1, s_2, and s_3. They are locked in this pattern:

Task A: $\ldots lock(s_1) \ldots unlock(s_1) \ldots$

Task B: $\ldots lock(s_2) \ldots lock(s_3) \ldots unlock(s_3) \ldots unlock(s_2) \ldots$

Task C: $\ldots lock(s_3) \ldots lock(s_2) \ldots unlock(s_2) \ldots unlock(s_3) \ldots$

The ceilings of the semaphores are as follows: s_1 and s_2 have ceiling priority M, and s_0 has ceiling priority H.

At time t_0 task C starts running, and tries to lock s_3: the priority of task C is higher than the ceilings of semaphores currently locked (there aren't any), and so the attempt to lock s_3 succeeds.

At time t_1, task B is invoked and (because it's priority is higher) pre-empts task C.

At time t_2, a little later, task B tries to lock s_2. This time, there is one semaphore already locked: s_3, which has a ceiling of M. Task B has priority M, which is not higher. So task B cannot lock the semaphore, and is blocked. Task C now inherits the priority of task B (priority M).

At time t_3, task A is initiated, pre-empts task C (since the task has a higher priority than the inherited priority C currently has), and begins executing. A little later, task A tries to lock semaphore s_1. There's only one semaphore that's locked: s_3, which has a ceiling of M. The priority of task A is higher than this, so it is allowed to lock semaphore s_1. Task A carries on executing.

At time t_4, task A has unlocked s_1, and finished executing. Task C carries on running now (since task B is blocked, and C is the highest priority runnable task). Task C then tries to lock semaphore s_2. The priority of task C is higher than the ceilings of all semaphores the semaphores locked by other tasks (there aren't any), and so the lock is permitted. Task C can then carries on executing.

At time t_5, task C finishes with s_2, and releases it, and carries on executing.

At time t_6, task C has finished with s_3, and drops back to its original priority (priority L). Now, task B can lock s_2. It can then go on to lock s_3, execute, release s_3, execute some more, and then release s_2.

At time t_7, task B finishes. Task C is now the highest priority task, and can continue, finishing at time t_8.

The described scenario is illustrated in Figure 4.4.

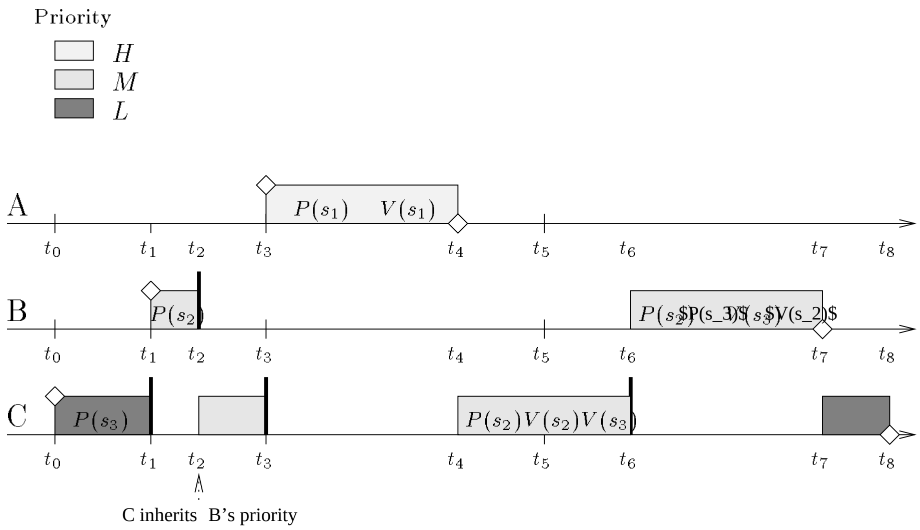

Figure 4.4: The Priority Ceiling Protocol (PCP).

Notice how task A is never blocked: its priority is higher than the highest ceiling of semaphores used by tasks B and C. Notice also how the blocking time of C is zero, since it's never delayed when trying to lock a semaphore.

Let's put some numbers to the example, and show how the theory predicts the worst-case blocking for each task. Task C holds semaphore s_2 for 10ms, and s_3 for 25ms (remember that the critical section for s_2 is nested inside the critical section for s_3, so the critical section for s_3 must be longer than the critical section for s_2). Task B holds s_3 for 5ms, and s_2 for 10ms. Task A holds s_1 for 5ms.

To find the blocking factor for task A, B_A, we look at all the lower priority tasks (B and C) and then look at all the semaphores they can access (s_2 and s_3). Then we look at only those that have ceilings with a priority equal to or higher than A: there are none. So the B_A is zero.

To find B_B, we look at all the lower priority tasks (just task C) and then look at all the semaphores they can access (s_2 and s_3). Then we look at only those that have ceilings with a priority equal to or higher than B: both s_2 and s_3 have ceilings equal to the priority of B, so they both count. The longest computation time a lower priority task executes while holding s_2 is 10ms, and the longest executing time while holding s_3 is 25ms. The largest of these is 25ms, and so B_B is 25ms.

Finally, to find B_C, we look at all the lower priority tasks, but there aren't any. So B_C is zero.

Now let's look at a bigger example. Look at the following table:

Task Name	T	D	C	priority
A	250	50	14	1
B	500	200	50	2
C	800	400	90	3
D	800	800	20	4
E	1000	1000	50	5
F	2000	2000	10	6
G	2000	2000	10	7
H	2000	2000	30	8

In the system, there are five semaphores: s_1, s_2, s_3, s_4, and s_5. The following table shows how they are locked:

Semaphore	Locked by	Time Held
s_2	D	3
s_4	D	3
s_1	D	9
s_2	H	13
s_3	E	4
s_3	B	4
s_4	A	1
s_5	F	7
s_5	H	7

We can work out the ceilings of the semaphores very easily:

Semaphore	Ceiling
s_1	4
s_2	4
s_3	2
s_4	1
s_5	6

Now it is straightforward to work out the blocking factors. Let's take the blocking factor for task D as an example. Tasks E, F, G, and H are lower priority tasks. These lower priority tasks together can lock semaphores s_2, s_3, and s_5. Of these s_2 and s_3 have ceilings with priorities higher than or equal to the priority of task D. Semaphore s_2 can be held by task H for 13ms. Semaphore s_3 can be held by task E for 4ms. The largest of these is 13ms, so the blocking factor of task D, B_D, is 13ms. We can easily calculate the blocking factors for the rest of the tasks, and can then calculate the worst-case response times. The following table gives the full results:

Task Name	B	R
A	3	17
B	4	68
C	4	158
D	13	187
E	13	237
F	13	247
G	13	271
H	0	288

Of course, it's very boring to have to do these calculations by hand, so it's best to write a simple program to do it instead.

4.7 The Immediate Inheritance Protocol

The priority ceiling protocol may be a little hard to understand. It's also a little difficult to implement in practice: the scheduling must keep track of which task is blocked on which semaphore, and what the inherited priorities are. It's also quite time consuming for the scheduler to have to work out whether a task can lock a semaphore or not.

It turns out that there is a simple protocol which has the same worst-case timing behaviour. It's generally called the *immediate inheritance protocol.* It has the same model as the priority ceiling protocol (eg. pyramid locking patterns, no holding of semaphores between invokations, etc.), but has a different run-time behaviour.

It's run-time behaviour is this: when a task i wants to lock a semaphore s, the task *immediately* sets its priority to the maximum of its current priority and the ceiling priority of s. When the task finishes with s, it sets its priority back to what it was before.

That's it!

It's easy to see why a task i is only delayed at most once by a lower priority task (just like with the priority ceiling protocol): there cannot have been two lower priority tasks that locked two semaphores with ceilings higher than the priority of task i. This is because one of them will have instantly inherited a higher priority first. Because it inherits a higher priority, the other task cannot then run and lock a second semaphore.

It turns out that we don't actually need to lock or unlock s. The reason is simple: the semaphore s cannot have been locked when task i comes to lock it, because otherwise there would be another task running with the same priority as i, and task i wouldn't be running. If task i wasn't running, then it couldn't execute code to try lock s in the first place!

Because the inheritance is immediate, task i is blocked, if at all, *before it starts running.* This is because, if a lower priority task holds a semaphore that can block task i it will be running at a priority at least as high as task i. So when task i is invoked it won't start running until the lower priority task has finished. At that point, there are no semaphores held that could block i, and so task i can run freely (unless preempted by a higher priority task, of course!). This property of the immediate inheritance protocol proves very useful when we come to implement a scheduler (as we will see later).

In many operating systems, concurrency is controlled by disabling interrupts (many processors have prioritised interrupt handlers, too; if you look through the source code to Unix, you'll see C procedure calls like `spl6()` (which stands for 'set priority level 6'): these are mapped to machine code instructions disabling all interrupts with priorities less than or equal to 6 (say). This protocol works just like that, only instead of interrupt handlers we have tasks. Because of this similarity, it is very easy to make interrupt handlers appear just like tasks: if a normal task accesses data shared with an interrupt handler, the task would execute a `SET_PRIORITY` call to the scheduler. In the scheduler, the top few priorities are mapped to processor interrupt priority levels. If the call to the scheduler said to set the priority of the calling task to one of these priority levels, the scheduler would disable processor interrupts.

Because the worst-case timing performance of this protocol is the same, the analysis just developed (ie. the calculation of blocking factors) remains unchanged for this protocol.

For all these reasons, we should use the immediate inheritance protocol instead of the priority ceiling protocol.

Chapter 5

The Jitter Problem

5.1 Introduction

In this section we are going to start dealing with some real-world behaviours of real-time tasks. There's more scheduling analysis to develop if we're to do this, but it's worth it: at the end of this chapter we'll have laid the foundations for modeling things like real-world scheduling overheads, variable latencies in event processing, and handling network messages.

5.2 Release Jitter

So far we have talked about 'invoking' tasks. We need to be a little more precise about the word *invocation*: we say that a task is invoked if it is possible for the scheduler to make it runnable. The scheduler makes a task runnable by *releasing* it. A task i is *released* when the following is true: if there are no tasks of higher priority than i that are ready to run then the processor starts to execute task i. Notice how this is different to 'invoked': if a task is invoked but not released then task i will not be run, and the processor could become idle. If a task is invoked and released then task i will start to execute whenever higher priority tasks have finished, and the processor cannot become idle until task i has finished. We've deliberately tried to avoid defining 'release' in terms of manipulating priority queues of tasks, changing task states to 'runnable', and so on. It's important to keep the implementation realities away from the scheduling model at this stage (we'll see why later in this chapter).

In previous chapters we've assumed that there is a minimum time T_j between successive invocations of a task i, and that the task is released as soon as it is invoked. But what if there is a variable (but bounded) delay between invocation and release? Even if the invocations of a task are made at regular periods, the release of the task could 'jitter' about. We call this *release jitter*. We can define it like this:

> *Release jitter is the difference between the earliest and latest time a task could be released relative to the invocation of the task*

It turns out that our original equations don't handle release jitter: a task may be invoked with time T between invocations, but it is not guaranteed to be *released* with a minimum of T between invocations. Consider the following example:

We have two tasks: a high priority task H, and a low priority task L:

Task Name	T	D	C	R
H	30	20	10	10
L	1000	25	15	25

By our existing analysis, these tasks are schedulable. But, as illustrated in Figure 5.1 and explained below, task L might miss its deadlines if jitter is introduced.

- At time t task H is *invoked*. But there is a delay of 10ms before the scheduler *releases* H (perhaps the task is invoked by a message sent from one processor to another, with a variable delay of up to 10ms for the message to get to its destination). So task H is released at time $t + 10$.
- At time $t + 10$ the low priority task L is *released*, but it is preempted immediately (because the high priority task has just been released too).

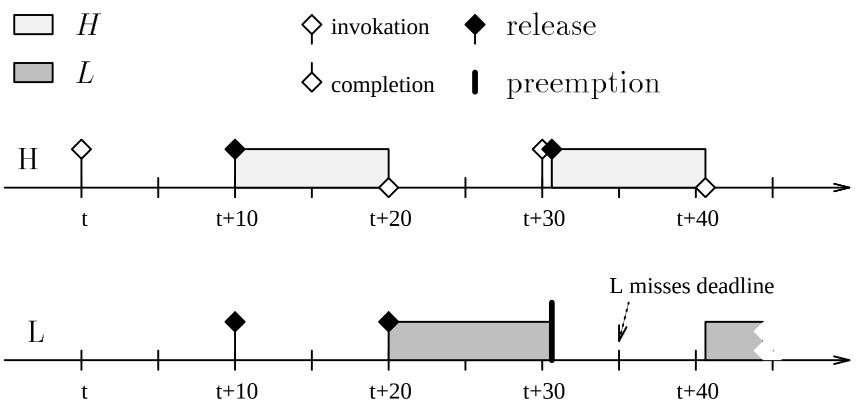

Figure 5.1: The effects of jitter.

- At time $t + 30$ the high priority task is invoked again (one period later), but this time the delay between invocation and release is just 1ms. So task H is released at time $t+31$. By this time, task L has completed 11ms of execution. Task H preempts task L and starts running.
- At time $t+41$ task H completes executing. But task L has already missed its deadline (its deadline was 25ms after it was invoked, at time $t + 35$).

Task H has experienced release jitter, and caused task L to miss its deadline, even though our existing analysis said this wouldn't happen. This is because there was just 21ms between two successive releases of task H; the analysis assumes 30ms (ie. T_H).

Notice how if the delay between invocation and release was *constant* then task H would experience no release jitter.

We must update the analysis if we want to allow tasks to experience release jitter. Let's do that now. Call $J_i^{biggest}$ the longest delay from invocation of task i to release of task i. Call $J_i^{smallest}$ the smallest delay from invocation to release. The release jitter of task i we'll denote as J_i Remember from earlier that this is the difference between the longest and shortest delays from invocation to release? Then J_i is given by:

$$J_i = J_i^{biggest} - J_i^{smallest} \tag{5.1}$$

We can see that if $J_i^{biggest} = J_i^{smallest}$ then the delay is constant, and we have no release jitter.

Going back to the example we see that J_H was 9 (the biggest delay was 10, and the smallest delay was 1). In the example, task L is preempted twice if $R_L > 21$. So we say that two preemptions of task L may occur if:

$$R_L > T_H - J_H \tag{5.2}$$

Task L may be preempted *three* times if:

$$R_L > 2T_H - J_H \tag{5.3}$$

(you can see this by ignoring the computation time required by task L in the example and keep extending the response time).

Task L may be preempted *four* times if:

$$R_L > 3T_H - J_H \tag{5.4}$$

We can generalise this and say that task L may be preempted n times if:

$$R_L > (n-1)\, T_H - J_H \tag{5.5}$$

To work out the maximum number of preemptions of task i we need to find the maximum value of n, given R_L, T_H, and J_H. We re-write the above as:

$$\frac{R_L + J_H}{T_H} > (n-1) \tag{5.6}$$

The largest value of $n-1$ that satisfies the above condition is (by definition of the ceiling function):

$$n - 1 = \left\lceil \frac{R_L + J_H}{T_H} \right\rceil - 1 \tag{5.7}$$

And so the largest value of n satisfying the condition is given by:

$$\left\lceil \frac{R_L + J_H}{T_H} \right\rceil \tag{5.8}$$

This is the largest number of preemptions of a lower priority task L by a higher priority task H. This is just what we need to update our existing analysis to include release jitter:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{5.9}$$

That would seem to be it. Except that we have a slight problem with where to measure the worst-case response time from. Should the worst-case response time R_i *include* the delay between invocation and release? (ie. should we measure the response time relative to the *invocation* of the task, or the *release* of the task?). Well, the most sensible way would be to measure it from the invocation, and so we should re-write our equation like this:

$$w_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \tag{5.10}$$

Where w_i is the time taken for the task to complete once it has been released. The notation 'w' is used because the time the task is running for is a time 'window', during which higher priority tasks can be released and preempt.

The worst-case scenario for a task i in a situation with three higher priority tasks is illustrated in Figure 5.2. In the figure, task i is invoked at time t and released at time $t + J_i^{biggest}$. Exactly at that time all higher priority tasks (j, k, and l) are released. These tasks are also suffering their worst possible delay from invokation. Now, the maximum number of preemptions from release at time $t + J_i^{biggest}$ to completion at time $t + R_i$ will occur, if for each higher priority task, the last release before $t + R_i$ occur as early as possible. If we use t'_m to denote the time of the last invokation of task m before $t + R_i$, then the corresponding earliest possible release will be at $t'_m + r_m^{smallest}$. What we get is an execution with a maximum number of releases of higher priority tasks during the critical window w_i, i.e. the time from $t + J_i^{biggest}$ to $t + R_i$.

The worst-case response time for a task i is now given by:

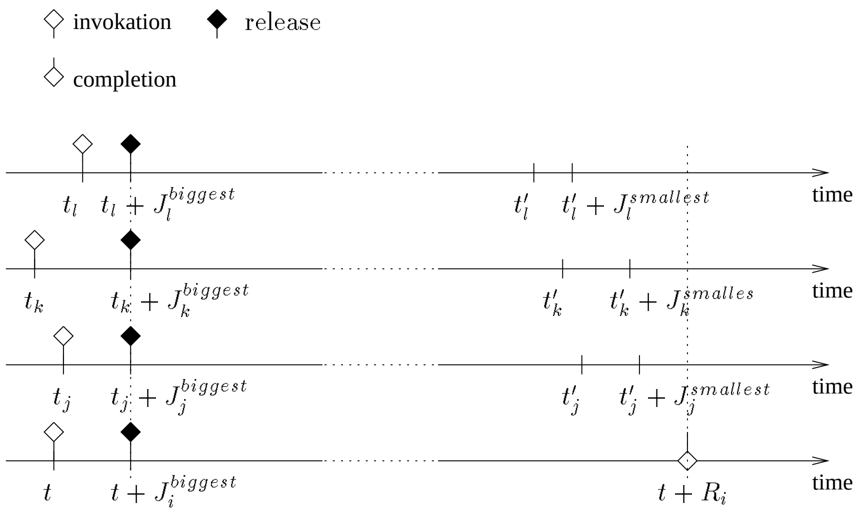

Figure 5.2: A worst case scenario with release jitter.

$$R_i = J_i^{biggest} + w_i \tag{5.11}$$

This can be compared to the deadline D_i as before. But there's a slight problem: because we have changed where we measure the response time from, and hence where we measure the deadline from, the optimal priority ordering is no longer *deadline monotonic*. Instead, we must order our tasks in $D - J^{biggest}$ order. Later we'll see that there are some cases where even this modified rule breaks down. To fix the whole problem we'll give an optimal algorithm to set priorities.

Note that we probably still want to restrict the deadline of a task to be less than the period of the task to make sure that at any time only one invocation of the task is runnable (we will see later in this book when we look at real-time communication that we need to make this restriction for real time messages).

5.3 Polling and Release Jitter

We talked in the previous section about one cause of release jitter was variability in network traffic: a message might be queued periodically with no jitter, but because of scheduling on a shared network, it might take a variable time to arrive at the destination. If the scheduler at the destination releases a task to handle the message, then the task would suffer release jitter (we'll look more at this aspect of real-time systems later in this book when we come to see how to do real-time networking and real-time distributed systems). But there are other causes of release jitter in fixed priority scheduling, one of which is *polling*.

We're going to look at at release jitter caused by polling in this section. It's important to do this because polling is a common way of handling events in the environment. Also, it's common to implement fixed priority scheduling by polling (later in this book we will look at implementing fixed priority scheduling and it will help to already know about the effects of polling). What we're going to do in this section is build up a model of how polling works and see how the current analysis fits.

Let's say that when we poll we'll do it with a periodic task (let's call it task *poll*) checking to see if an event has occurred. If it has occurred then we can say that a second task (call it task *work*) is released to handle the event. Of course, this *model* for polling and the exact *implementation* in reality might be different: we might use a single task which merely executes an 'if' statement, with a call to a procedure to do some work. But this implementation is compatible with the model (the single real task is modelled as two tasks with adjacent priorities, with the 'work' task having the lower priority).

Clearly, the rate at which task *work* is run is the same as the rate of the event (let's say that the event has a minimum period - ie. a maximum rate - denoted T_{event}).

So task *work* would be like this:

Attribute	Description
C_{work}	Computation time required to process event
T_{work}	Period of task *work*, equal to T_{event}

We still have a question though: "what is the jitter that the second task 'inherits' from the polling task?" To answer this, look at the following diagram:

COMMENT: diagram here showing polling

We've made the scenario very simple: the polling task takes zero time to poll for the event, and takes zero time to release the work task. This isn't very realistic[1] but we'll fix that problem soon.

At time t the poller checks to see if the event has occurred, but it hasn't yet, so nothing is done. Then at time $t + a\ bit$, the event occurs, but the poller doesn't see the event until the next poll: at time $t + T_{poll}$. When the poller does see the event, task *work* is released and processes the event.

Task *work* was logically invoked as soon as the event occurred (time $t+a\ bit$), but wasn't released until some time later ($t + T_{poll}$) because of the polling delay. So $J_{work}^{biggest}$ is equal to T_{poll} (it's safe to make things simple and assume that "$a\ bit$" is zero). It's also safe to make the assumption that zero is the shortest time between the event occurring and task *work* being released, ie. $J_{work}^{smallest}$ is 0. From this we say that J_{work} is equal to T_{poll}. And so we have all the attributes of task *work*, and we can work out the worst-case response times of all the tasks. From this we see that the 'interference' from task *work* on a lower priority task i is bounded by:

$$\left\lceil \frac{w_i + T_{poll}}{T_{work}} \right\rceil C_{work} \tag{5.12}$$

So that's the end of the story? Well, not quite. We cheated a bit when we drew the diagram earlier. The period of the event on the diagram (T_{event}) was chosen as a particularly nasty one: it has a period equal to $2T_{poll} - a\ bit$, which is just the right value to maximise the jitter. What would have happened if we had picked a value of $2T_{poll}$ instead? Let's re-draw the diagram:

COMMENT: new diagram

[1]Actually it might be realistic! Often hardware interrupts are implemented like this (between every instruction - which on some RISC processors is always one clock cycle - the hardware checks to see if an interrupt occurs, and jumps to the handler if it has)

See how task *work* is now released with no jitter? $J_{work}^{biggest}$ is equal to T_{poll}, but $J_{work}^{smallest}$ is too. This means that the release jitter J_{work} is equal to zero. That's nice: it means that if we poll for something and the periods are just the right values then there is no jitter (there is, of course, a polling delay for task *work*: it's worst-case response time is equal $T_{poll} + w_{work}$). Remember that the 'event period' is a *minimum* time between events occurring: if we stretch the time between the events a little bit further, it's never worse than what happens with an event period of $2T_{poll}$.

So how do we take advantage of this nice behaviour where the release jitter is removed? Well, it turns out that the 'interference' from task *work* on a lower priority task i can be alternatively bounded by:

$$\left\lceil \frac{\left\lceil \frac{w_i}{T_{poll}} \right\rceil T_{poll}}{T_{work}} \right\rceil C_{work} \tag{5.13}$$

It's possible to prove that this equation never results in a bound greater than the simpler one given earlier. It's also possible to prove that when T_{event} is a whole-number multiple of T_{poll} that the interference is the simple term from our original analysis:

$$\left\lceil \frac{w_i}{T_{event}} \right\rceil C_{work} \tag{5.14}$$

This is the same as saying that there are no jitter effects from the polling. Of course, in the tradition of the best textbooks, we say that these proofs are left as an exercise for the reader!

We promised earlier to come back to the problem of making unrealistic assumptions about the overheads of the polling task in our model. We'll deal with these assumptions now. The polling task is just like any other task: it has a worst-case computation time C_{poll}, a period T_{poll}, an invocation delay of between $J_{poll}^{smallest}$ and $J_{poll}^{biggest}$, a release jitter of J_{poll}, and a worst-case response time R_{poll}. What happens to our model and analysis if $J_{poll}^{biggest}$ or C_{poll} are non-zero? If we make the assumption that the polling task has a higher priority than task *work* then the surprising answer is "not much". Let's see what needs to change.

Task *work* is activated by the polling task (probably by some 'activate' call made to an operating system, or something similar) when it's been running for a bit and has 'seen' the event. In the call, the operating system marks the task *work* as runnable (updating a list of runnable tasks, or whatever), and then switches back to the poller (because it's of higher priority than the task just activated). Task *work* only gets to run after the polling task has finished[2].

A question: at what time is task *work* 'released'? When the call to the operating system is made? When the operating system call finishes? When the polling task finishes? The answer is "none of these". From the view of the tasks, it makes no difference *when* in the execution of the the polling task the call to the operating system is made: it could be made right at the start or right at the end. The timing behaviour would be the same. Task *work* would still be released when the polling task finished executing.

This is an important observation, because it means that we can say that **task *work* is released at the same time the polling task is released**. This is why at the start of this chapter we carefully defined 'release': if the polling task were to complete its work and start executing then the processor would not become idle (task *work* would start running). It's useful for the release time to be defined like this: the analysis we've derived so far in this section won't change much. Besides, if we said that that task *work* is released only when the call to the operating system has completed then the analysis would over-estimate the worst-case response time. The following example shows why:

COMMENT: example with two pictures to show two ways of treating release time

It's important to remember, though, that the updated analysis only holds if the polling task has a higher priority than any task it releases. If we allowed the polling task to have a lower priority then we get lots of priority inversion. This is because any task of higher priority than the polling task, but lower priority than the released task, could delay the release of the higher priority task (think about it for a moment). In practice, this isn't a problem: if the deadline for processing the event is short then the period (and deadline) of the polling task is even shorter. Priorities should reflect deadlines, and so the polling task should have a higher priority anyway.

[2]And, of course, after all other tasks with priorities between the poller and task *work* are also finished

5.4 Techniques for controlling Jitter

COMMENT: Dual priority and offsets

Chapter 6

Programming Priority-based Systems

Chapter 7

Implementing Fixed Priority Schedulers

7.1 Introduction

We mentioned several times in previous chapters how the scheduling equations we have developed so far do not allow for the behaviour of an implemented scheduler. For example, the equations assumed zero scheduling costs. In this chapter we will describe two common ways of implementing a fixed priority scheduling and show how the analysis can be changed to account for the behaviour of these implementations. First, though, we need to talk about *context switches*.

7.2 Context Switch Overheads

The *context* of a task is the set of registers that the task uses. On a processor like the 68000, there may be many registers: temporary data registers, the program counter, the stack pointer, the status register, and so on. There may also be other registers which control the memory management hardware (mapping logical memory addresses to physical addresses in RAM), the floating point unit, *etc.* When the scheduler wants to switch from running one task to running another task, it must save the context of the old task, and load the context of the new task. The place where the context of a task is saved is usually called the *task control block*, and the save and load process is known as a *context switch*.

The context switch can take some time to execute (try writing one for the 68000 and count the number of cycles), and there can be a lot of context switches over the execution of a task. These overheads can mount up and be quite considerable, so it is important to bound the execution time due to context switches.

Every preemption of one task by another task could results in *two* context switches: one switch from the lower priority task to the preempting task, and one switch back again when the preempting task completes. So if a higher priority task j preempts a task i five times, there can be ten context switches due to that task. But we know how many times a high priority task can preempt a lower priority one: our existing analysis tells us this, and uses it to work out the total time a high priority task can execute, delaying a low priority task. So, to allow for the context switches we can add to the computation time of each task the time taken for two context switches. If we denote the context switch time as C_{sw} then we can update the analysis we have so far:

$$R_i = J_i + w_i \tag{7.1}$$

(Remember that we choose to measure the worst-case response time from when the task is invoked, and not when it is released: the release jitter J_i is added on to the worst-case response time).

Recall that the term w_i represents the worst-case "queuing delay": the time between releasing the task in to the priority ordered run queue, and the time when the task completes its execution. The term is given by:

$$w_i = C_i + 2C_{sw} + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil (C_j + 2C_{sw}) \tag{7.2}$$

So we have accounted for the computation time costs of context switches. But there is one more job to do: the code for a context switch is usually not preemptable, and so one task cannot preempt another while a context switch is going on. To allow for this delay we must make sure that the blocking factor takes account of this. One simple way to do this is to make sure that for all tasks the blocking factor B is at least C_{sw}.

7.3 Implementation: Event Scheduling

We have yet to talk about the details of how the scheduler is implemented (apart from talking about context switches and blocking in the kernel). In this section we will talk about one way of implementing a fixed priority scheduler: the *event scheduling* approach.

We create two queues of tasks in the scheduler: one is called the *run queue*, and is ordered by task priority. The other is called the *delay queue*, and is ordered by the time a task wants to next be released. When a task executes a `DELAY_UNTIL` call, the scheduler takes the task from the run queue, and places it into the delay queue, ordered by the parameter of the `DELAY_UNTIL` call. When the scheduler has moved the task between the two queues, it sets up a hardware timing chip to interrupt the CPU at the release time of the task at the head of the delay queue. The scheduler then context switches to the task at the head of the run queue.

When the interrupt from the timer chip occurs, all the tasks in the delay queue with release times before or the same as the interrupt time will be moved to the run queue. The scheduler sets up the timer chip for the next interrupt, and then switches to the highest priority task.

As we mentioned before, the delay for the scheduler can be accounted for by making sure the blocking factor B for all tasks includes the time for the `DELAY_UNTIL` call. The time taken to process the call is accounted for in the worst-case execution time of the task making the call. But there is other computation in the scheduler not accounted for: the time processing the timer interrupts.

How can we allow for the computation from timer interrupts? A simple way to do this is to realise that an interrupt occurs at the same time as a task wants to run. For each real task in the system, we can create a "fictitious" task: each has a very high priority, the same period as the real task it models, and an execution time equal to the worst-case computation time required by the interrupt handler.

The following equation updates our analysis to handle this model:

$$R_i = J_i + w_i \tag{7.3}$$

(as we had before)

$$w_i = C_i + 2C_{sw} + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil (C_j + 2C_{sw}) \; + \tag{7.4}$$

$$\sum_{\forall j \in alltasks} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_{timer}$$

Where C_{timer} is the time taken to respond to the timer interrupt (ie. to move a task from the delay queue to the run queue). If a new task becomes the highest priority task, a context switch is required. The time for this is accounted for with $2C_{sw}$ (as we talked about earlier). $alltasks$ is the set of all tasks in the system (including task i).

This equation shows that the scheduler causes *priority inversion*: a task of lower priority than i shows up in the set $alltasks$. This is because the timer interrupt occurs at high priority, even though the interrupt handler may do work that is on behalf of a low priority task.

There are some awkward problems to overcome when using event scheduling: the timer hardware must be quite sophisticated. The scheduler needs to program the timer to interrupt at some future time, which requires that the timer be able to store long times. On most processor boards, the timer is not big enough to store large durations, and will often 'wrap' around. This causes more complexity in the scheduler: when the timer wraps this must be detected by the scheduler so that it can keep track of the current time. To do this an interrupt must be generated when the timer wraps, causing more scheduling overheads (these can be modelled easily by creating a fictitious task with a period equal to the maximum duration of the timer). Another problem with this timer is the *race condition* problem: the scheduler may instruct the timer to interrupt at a time so near in the future that by the time the scheduler has finished the context switch to the new task, the timer interrupt may have already occurred again. With some pieces of hardware an interrupt may be lost if it occurs while a previous interrupt from the same device is still being serviced.

Because of this complexity in implementation, a simpler (but less efficient) approach was taken: *tick driven scheduling*.

7.4 Tick Driven Scheduling

Tick driven scheduling is a simpler approach to implementing fixed priority scheduling. As before, the scheduler can be invoked by a task making a `DELAY_UNTIL` call, and as before the scheduler makes context switches whenever a new task becomes the highest priority runnable task. However, unlike the event scheduling approach, the timer interrupt is not programmed to interrupt at a specific time. Instead, the timer hardware is programmed to raise an interrupt to invoke the scheduler at a regular interval, called the tick period T_{tick}. When the scheduler is invoked by the timer interrupt, it merely *polls* the delay queue to see if there are any tasks to be made runnable (ie. to be inserted in to the priority-ordered run queue).

This removes the problems with the event scheduling approach: there are no race conditions or lost interrupts as long as the scheduler takes no longer than T_{tick} to finish. But the analysis for event scheduling is no longer sufficient: because the scheduler polls for the release of tasks, they may be delayed by up to a polling period. This delay is not accounted for in the existing analysis, and so we must update the analysis.

We talked in previous chapters about how a task might be delayed from being logically invoked to actually being released: we called this problem *release jitter*. We have a similar problem here with tick scheduling: a task may be released as soon as it is invoked (for example, if it is invoked just as the scheduler starts to run), or it may be delayed for some time (for example, if it is invoked just *after* the scheduler starts to run). So we can see that a given task may be delayed from invokation to release by between zero and T_{tick}. We can account for this delay by saying that the task *inherits* an additional release jitter from the tick scheduler of T_{tick}, the tick period.

We must also account for the time the tick scheduler takes to run. We could do this by finding out the worst-case execution time of the scheduler, and then treating it as a simple high-priority periodic task with period T_{tick}. In the worst-case, the scheduler must move all the tasks from the delay queue to the run queue, taking some time. But this situation occurs only rarely (in fact, only when all tasks are due for release at the same time), and so this approach to modelling the overheads would be very pessimistic.

A better thing to do is to create a set of "fictitious" tasks (just like we did for event scheduling) to model the queue manipulation costs, and to create a simple periodic task with period T_{tick} to model the costs of the timer interrupt.

Because the tick scheduling approach leads to inherited release jitter for tasks, we must make sure that the ficitious tasks also inherit the same release jitter. So, we can put together these things to update our simple analysis to take account of the behaviour of a tick scheduler:

$$R_i = J_i + T_{tick} + w_i \tag{7.5}$$

(notice how T_{tick} adds to any release jitter task i may already have had)

$$w_i = C_i + 2C_{sw} + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j + T_{tick}}{T_j} \right\rceil (C_j + 2C_{sw}) + \tag{7.6}$$
$$\sum_{\forall j \in alltasks} \left\lceil \frac{w_i + J_j + T_{tick}}{T_j} \right\rceil C_{queue} + \left\lceil \frac{w_i}{T_{tick}} \right\rceil C_{tick}$$

where C_{queue} is the time taken to move a task from the delay queue to the run queue, and C_{tick} is the time taken to handle the tick interrupt (saving registers on the stack, *etc.*).

Notice how T_{tick} adds to the release jitter of both the real tasks and the fictitious tasks. Also notice how the interrupt handling costs are modelled. Also, just as we had with the event scheduling, the longest time the scheduler is non-preemptable counts towards the calculation of B_i.

Finally we discuss the inherited release jitter again: if a task is a strictly periodic task, and has a period that is an integer multiple of T_{tick}, and is defined to be invoked when the tick scheduler is invoked, then the task does not inherit any extra release jitter. This is because when the scheduler is invoked, the task has only just been invoked itself, and is then immediately released: this means that (by definition) it wouldn't suffer any release jitter. So the term $J_j + T_{tick}$ could be removed from the above equation. This is useful because it means that by carefully choosing the value of T_{tick} we can reduce the delays a task experiences, and make that system easier to show to be schedulable.

To summarise: we have discussed how to implement fixed priority scheduling with a tick approach, and have updated the analysis to take account of its behaviour.

7.5 Comparing the Two Scheduling Models

We can now compare the two scheduling models and see what advantages and disadvantages each has.

An important advantage of the tick scheduling approach is that it is *simpler*: this means that it is easier to write, and easier to understand. Also, it does not require so much hardware support as event scheduling: remember that event scheduling requires a timer that we can program to 'interrupt at' some time. Even if we do have this hardware, we must be careful how we handle timer overflows: to measure a long interval of time, most timers will wrap around, and we must keep track of how many times this happens (otherwise we cannot keep track of the current time). Event scheduling can also suffer from race conditions: where the next event can occur before the current event has been dealt with (with some hardware we might lose interrupts because of this).

Event scheduling is generally more efficient than tick scheduling: we can see from the equations that in general the tick approach leads to release jitter, and hence delay. This affects the real-time performance of the system. We have mentioned how some tasks might not inherit release jitter, and if most of the system is made up of tasks like this then the difference may not be so great. Because we now have analysis to describe the impact of the scheduling approach, we can make a quantitative decision for a given system about which of the two approaches to use. This is especially useful when we come to choose a value of T_{tick}: too large a value, and the release jitter (and hence worst-case response time) of a task becomes too big. Too small, and the overheads from timer interrupts become too large.

7.6 Summary

We saw how to account for the timing behaviour of scheduling implementations. The scheduling overheads are made up from the context switch costs, timer interrupt handling costs, and the costs of manipulating the delay and run queues. For tick scheduling, there is the additional impact of inherited release jitter. The event scheduling approach is more efficient than the tick scheduling approach, but tick scheduling is easier to implement and requires less hardware support.

Chapter 8

Mixing Hard and Soft Real-Time Processing

Chapter 9

Real-Time Networking

9.1 Introduction

When we talk about distributed real-time systems there are lots of issues to consider, but the most important of these is *communication*. In a distributed system, by definition, a number of processors communicate results via some media. In order to reason about *end-to-end* timing behaviour – where we wish to talk about the longest time between an event on one processor and some resulting action on another – we must not only be able to talk about the timing behaviour of the processors, but also the timing behaviour of messages sent on the communications media. The goal of this chapter is to be able to predict the longest time between a message being queued and the message reaching its destination.

A lot of work has been done over the years in developing and dealing with real-time communications. Many different communications protocols have been proposed, and people have tried to show how each can be used to give timing guarantees for real-time messages. In this chapter we will be talking about one of these protocols, and show how we can get the timing guarantees we need. This network protocol is called *Controller Area Network*, or CAN for short.

CAN was first thought about in the late 1970's in the car industry: there is a problem when trying to connect together a number of processors over a cheap shared serial bus, and CAN was designed to overcome this. The first problem is to find a protocol which has a good real-time performance (where urgent messages are prioritised over less urgent messages), and which can be analysed to show worst-case timing behaviour.

With a shared communications media we have the problem of deciding who gets access to the media, and when. This is just like the processor scheduling problem (deciding who gets access to the processor), but with a difference: the decision making procedure cannot be centralised (like on a CPU). All the processors in the distributed system must agree on some protocol for access to the bus. This protocol must have good real-time performance, and must be analysable for the worst-case timing behaviour.

Before CAN, most real-time protocols were based on the idea of a *token*: a processor gets the right to transmit on the shared network when it receives a token. After it has finished transmitting (or, more usually, after some time limit) it must pass the token on to another processor. The token is passed around all the stations in a logical ring.

The real-time behaviour of token passing protocols aren't usually very good: this is because a processor with very urgent data to send must wait for the token to return, which can take a long time. It can take a long time because each other processor in the system is allowed to hold the token for some period of time, even if the processor has only non-urgent messages to send. In effect, an urgent message is delayed by the transmission of a lot of non-urgent messages: we called this *priority inversion* when we talked about processor scheduling. Priority inversion reduces the real-time performance of a protocol.

One approach that token passing protocols try to take is to make the time each processor can hold the token a short period of time. However, the side-effect of this is that the proportion of the bus utilisation used for passing the token go up, and so the available bandwidth for sending messages goes down. This also affects real-time performance.

The approach taken by CAN is different: it associates a priority with each message to be sent, and uses a special arbitration mechanism to ensure that the highest priority message is the one transmitted. The priority of a message is a unique static number, and can also be used to identify the message. This is why the priority is known as the *identifier*. Because CAN uses fixed priorities to arbitrate between messages, we should be able to use the results of our processor scheduling theory to work out the longest time taken to send a message: the goal of this chapter. In the rest of this chapter we will describe the CAN protocol in more detail, and then show how to analyse its worst-case timing behaviour.

9.2 The Basic CAN Protocol

CAN is shared broadcast bus, with limits on speed and length (these are very important to the protocol as we will see in a moment): at 1Mbit/sec the bus must be no longer than 50m; at 500Kbit/sec the bus must be no longer than 100m. The bus can have an arbitrary number of *stations* connected to the bus (a station is a processor plus a special controller which communications from the processor to the bus, and vice versa).

Each station has a queue of messages, ordered by priority, that it wants to send. While the bus is busy (*i.e.* there is a station already sending a message) the stations wait. As soon as the bus becomes idle, all the stations with messages to send enter into the CAN arbitration algorithm. Each station starts to transmit the priority of its highest priority message, from most significant bit to least significant bit.

They all start to transmit at the same time, and normally this would result in gobbledegook on the bus. But the bus has some interesting electrical properties: if any one station transmits a '0' bit then the state of the bus will be '0', regardless of what other stations try to transmit. The state of the bus can only be a '1' if *all* stations transmit a '1'. In fact, the bus behaves just like a giant AND-gate. Because of this behaviour, the '0's are called *dominant* bits, and the '1's are called *recessive* bits.

Each station monitors the bus during transmission. During arbitration, if a station transmits a '1' but monitors the bus and sees a '0' – a *collision* – then it knows that there was another station transmitting still. The arbitration protocol says that any station detecting a collision must *back off*: because messages are transmitted with the most significant bit first, the station detecting a collision does not contain the highest priority message in the system, and must wait until the next time the bus is idle before entering arbitration again.

Let's take an example. Suppose we have three stations, A, B, and C. Station A's highest priority message has a priority of 4 (100 in binary). Station B's highest priority message has a priority of 5 (101 in binary). Station C's highest priority message has a priority of 7 (111 in binary). After the idle period, all three stations start to transmit the message priorities, most-significant-bit first.

First, all three stations transmit '1' (a recessive bit), and all three stations read a '1'. Because all three stations read from the bus what they sent to the bus, no station backs off.

Next, stations A and B send a '0', and station C sends a '1'. Because at least one station has sent a dominant bit (a '0'), all three stations will read a '0'. Stations A and B read what they sent, and do not back off. However, station C sent a '1' but read a '0', and so knows that it does not contain the highest priority message. So it backs off (but will enter arbitration the next time the bus becomes idle).

Next, station A sends a '0', and station B sends a '1' (station C does not transmit anything because it has backed off). Both stations A and B will read a '0'. Station B detects a collision and backs off, leaving station A to carry on transmitting the rest of its message.

So the highest priority message – priority 4 – is transmitted on the bus. Notice how lower numbered messages are of higher priority (zero is the highest priority). Notice also how, from the view of station A, the message is transmitted without interruption: station A cannot tell if the bus was free of other stations, or if all stations tried to send data but failed.

A message is more than just its priority: after the priority has been transmitted, and arbitration finished, the rest of the message is transmitted. A CAN message contains several field: the priority (or *identifier* field) as we just mentioned. This consists of 11 bits. The field containing the data in the message is between 0 and 8 bytes long. Finally, there are some other trailing fields (a CRC field for checking the message has not been corrupted, and some other 'housekeeping' fields). In a CAN message, there are 47 bits of overhead and between 0 and 64 bits of data.

CAN has a special bit pattern as a marker used to signal errors to all stations on the bus. The marker pattern is six bits of the same sign in a row: 000000 or 111111. We must make sure that the bit format for a message doesn't contain this special marker by accident (for example, the data field might contain these values. We solve this by *bitstuffing*: whenever the CAN controller (the interface between the CPU and CAN bus) spots five bits in a row of the same sign, it inserts an extra bit of the opposite sign into the bitstream (for example, if the bit stream contains ...00000...then after bit stuffing it contains ...000001...). At the receiving controllers, the stuffed bits are stripped out. Of the 47 bits of overhead (identifier, CRC, *etc.*), 34 of them are are bitstuffed, along with the data field. So we might think that the number of stuff bits that might be inserted into the bitstream is given by:

$$\left\lfloor \frac{34 + 8n}{5} \right\rfloor \tag{9.1}$$

where n is the number of bytes in the message. This is however not correct. More stuff bits might be inserted since there are two complementary bit patterns that must be avoided. To see this consider the bit pattern:

$$1111\ 1000\ 0111\ 1000\ 0111\ 1 \tag{9.2}$$

which will be stuffed as follows (stuffed bits are boldfaced):

$$1111\ 1\mathbf{0}000\ 00\mathbf{1}1\ 111\mathbf{0}\ 0000\ \mathbf{1}111\ 1\mathbf{0} \tag{9.3}$$

i.e. there will be 5 stuff bits, but, according to the equation above the maximum number of stuff bits is $\lfloor \frac{21}{5} \rfloor = 4$. Instead of the above equation we should use

$$\left\lfloor \frac{34 + 8n - 1}{4} \right\rfloor \tag{9.4}$$

to calculate the maximum number of stuff bits. This equation gives for our small example an upper limit of $\lfloor \frac{20}{4} \rfloor = 5$ stuff bits. Of course, we might do better than this if we knew the identifier of the message, and knew something about which values the data part of the message could contain.

9.3 Basic CAN Timing Analysis

As we mentioned in the introduction, we ought to be able to apply the work we have done on processor scheduling to CAN scheduling. The first thing we must work out before we do this is to answer the question "How long does it take to transmit an n byte message?" As we mentioned in the previous section, the CAN protocol inserts stuff bits into the bitstream. The total number of bits in a CAN message before bitstuffing is:

$$8n + 47 \tag{9.5}$$

Therefore the total number of bits after bitstuffing can be no more than:

$$8n + 47 + \left\lfloor \frac{34 + 8n - 1}{4} \right\rfloor \tag{9.6}$$

Let τ_{bit} be the time taken to transmit a bit on the bus – the so-called *bit time*. The time taken to transmit a given message i is therefore:

$$C_i = \left(8s_i + 47 + \left\lfloor \frac{34 + 8s_i - 1}{4} \right\rfloor \right) \tau_{bit} \tag{9.7}$$

where s_i is the size of message i in bytes. Notice how this time is denoted C_i: it is analogous to the worst-case computation time of a task.

If we put $s_i = 8$ into the equation, and assume a bus speed of 1Mbit/sec ($\tau_{bit} = 1\ \mu$s), we get $C_i = 135\mu s$. This is a good figure to remember: the largest message takes 135 bit times to send.

If we want to apply the processor analysis to CAN bus, we have to adopt the same model: each message i has a periodicity T_i, a transmission time C_i, and a *queuing jitter* J_i (queuing jitter is just like release jitter for tasks, but refers to variations in the time when messages are queued, typically depending on varying response times for the task that queues the messages). The priorities of messages are unique (in fact, the CAN protocol requires unique identifiers). A message can also suffer from *blocking*: when a message is delayed by the transmission of a lower priority task (just like when a lower priority task delays a higher priority one). This blocking time is denoted B_i.

We have already established the transmission time C_i. The blocking time is easily established too: a higher priority message could be queued just after arbitration starts (and is too late to enter ino arbitration). The longest time until the next arbitration period is the time for the longest message to be sent. Without knowing details of all the low priority messages on the bus, we can say that this will be no more than 135 bit times. If we do know about all the lower priority messages, then we need to find the largest one sent and work out its transmission time. So, B_i is given by:

$$B_i = \max_{\forall k \in lp(i)} (C_i) \leq 135\tau_{bit} \tag{9.8}$$

Finally, we can take our existing equations for processor scheduling theory and apply them here:

$$R_i = J_i + w_i \tag{9.9}$$

$$w_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \tag{9.10}$$

That may seem like "end of story", but there's more: the equations above are contradictory about how the protocol behaves. In one part they assume that a lower priority message cannot be interrupted by a higher priority one once they start transmission (hence the calculation for B_i). In another part, they assume that message i *can* be interrupted by a higher priority message. This should be fixed, because otherwise the analysis is pessimistic.

A higher priority message j cannot preempt message i if j is queued after the first bit of i has been transmitted: this is because the arbitration has started, and any higher priority message must wait until the bus becomes idle. So, we need to find the longest time taken for the first bit of task i to be transmitted without interference from higher priority messages. This queuing time is given by:

$$w_i = \tau_{bit} + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \tag{9.11}$$

The worst-case response time of the message includes the queuing jitter, the queuing time (the longest time before the first bit of the message is sent without interference), and the time taken to send the rest of the message (the whole message minus the first bit):

$$R_i = J_i + w_i + C_i - \tau_{bit} \tag{9.12}$$

We can re-write the above two equations as:

$$R_i = J_i + w_i + C_i \tag{9.13}$$

where w_i is given by:

$$w_i = B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j + \tau_{bit}}{T_j} \right\rceil C_j \tag{9.14}$$

So there we have it: analysis for messages sent on CAN bus.

Chapter 10

Distributed Hard Real-Time Systems

10.1 Introduction

10.2 Attribute Inheritance

So we should be able to apply the analysis to any given distributed system, and work out end-to-end response times, right? Wrong! We still have some values in the equations to fill in: how is the queuing jitter of a message worked out?

The answer comes from the idea of *attribute inheritance*: a task is invoked by some event, and will queue a message. The message could be queued soon after the event (for example, if there were no higher priority tasks, and the execution time to queue the message was small). The message could be queued late after the event: when the worst-case scheduling scenario occurs. This difference between the shortest and longest queuing times is, of course, the queuing jitter. We can say that the shortest time is zero (we might be able to do better than this with more clever analysis, but zero is a safe assumption; remember that a small jitter is preferable). We can say that the longest time is the worst-case response time of the queuing task (again, we might be able to do better than this assumption). So, the queuing jitter of a message i is given by:

$$J_i = R_{send(i)} \tag{10.1}$$

where $send(i)$ is the task that sends message i.

But the reverse happens too: what if a task is released when a message arrives? If this happens, then the task *inherits* release jitter from the message – in just the same way as a message inherits queuing jitter from the sending task. The soonest a message i can reach the destination processor is 47 bit times (the smallest message is 47 bits long). The latest a message i can reach the destination processor is R_i. So the destination task, $dest(i)$, inherits a release jitter of:

$$J_{dest(i)} = R_i - 47\tau_{bit} \tag{10.2}$$

Now we are starting to get close to our goal of obtaining worst-case end-to-end response times. The worst-case response time of task $dest(i)$, $R_{dest(i)}$, is measured from $47\tau_{bit}$ after the event invoking task $send(i)$. So, the time $47\tau_{bit}+R_{dest(i)}$ gives the end-to-end time from the event invoking task $send(i)$ to the completion of the task $dest(i)$.

So, is that the end of this chapter? No, because there are one or two problems yet to solve. The first is to do with the overheads due to receiving a message from the CAN controller.

10.3 Message Processing Overheads

With most CAN controllers, when a message arrives at the controller, an interrupt is raised on the host processor. The processor must then copy the message from the controller into main memory (and can then process the message, including releasing any tasks waiting for the message). This interrupt handling takes time, and we've already seen how we've got to bound these overheads.

One way of bounding the overheads is to assume that messages arrive at their maximum rate (once every 47 bit times). This gives an additional interference factor of:

$$\left\lceil \frac{w_i}{47\tau_{bit}} \right\rceil C_{message} \tag{10.3}$$

where $C_{message}$ is the worst-case execution time of the interrupt handler dealing with messages.

But this can be pessimistic: we know that messages can't arrive so rapidly for a long period of time. We can do the same thing we did when modelling overheads for the scheduler: we can create a set of high priority fictitious tasks, with the worst-case execution time of the task equal to $C_{message}$. A fictitious task is kicked off by the arrival of a message, and so inherits a release jitter just as we described above. So, we can say that the overheads due to message arrival interrupts are:

$$\sum_{\forall j \in incoming} \left\lceil \frac{w_i + R_j - 47\tau_{bit}}{T_j} \right\rceil C_{message} \tag{10.4}$$

where *incoming* is the set of all messages coming in to a processor and raising "message arrived" interrupts.

We now have two bounds for the overheads, and can take the lower of the two bounds. So, the factor we would add to our task scheduling equations is:

$$min \left(\sum_{\forall j \in incoming} \left\lceil \frac{w_i + R_j - 47\tau_{bit}}{T_j} \right\rceil, \left\lceil \frac{w_i}{47\tau_{bit}} \right\rceil \right) C_{message} \tag{10.5}$$

10.4 The "Holistic Scheduling" Problem

There's one final problem we must solve before we can use the analysis we've developed: the so-called *holistic scheduling problem.*

The equations giving worst-case response times for tasks depend on the timing attributes of messages arriving at the processor (for example, the release jitter of a task kicked off by a message arrival is inherited from the worst-case response time of the message; also overheads from "message arrived" interrupts are dependent on the worst-case response times of incoming messages). So we cannot apply the processor scheduling analysis until we have applied the bus scheduling analysis. But equations giving worst-case response times for messages depend on the timing attributes of tasks queuing the messages (for example, the queuing jitter of a message queued by a task is inherited from the worst-case response time of the message). So we cannot apply the bus scheduling analysis until we have applied the processor scheduling analysis.

If we cannot apply the processor scheduling analysis until we have applied the bus scheduling analysis, and *vice versa*, then how can we apply the equations? Well, we had a similar problem when trying to find the worst-case response time of a task: we couldn't find R_i until we knew R_i because it appeared on both sides of the equation! We solved the problem then by iterating to a solution, and we can do exactly the same now.

The first step is to assume all the calculated timing attributes are zero. Then we can apply both sets of analysis (for processors and for the bus) to obtain new values. We can then reapply the analysis with these new values. The iteration carries on until the calculated values don't change.

Understanding the full complexity and beauty of holistic scheduling is best done via a simple example. Consider the simple distributed real-time system depicted in Figure 10.1.

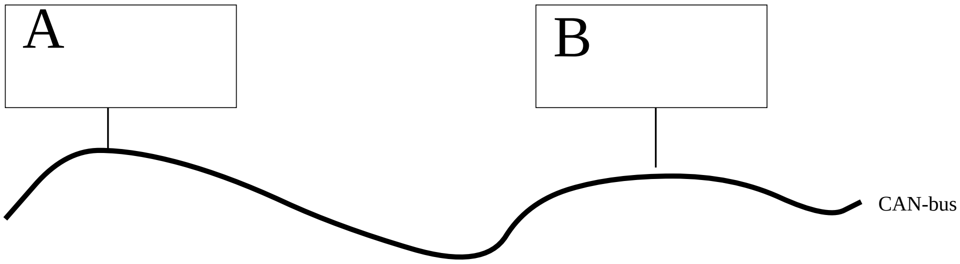

Figure 10.1: A simple distributed real-time system.

The three tasks TA_1, TA_2 and TA_3 are executing on node A and the tasks TB_1, TB_2 and TB_3 are executing on node B. There are two semaphores guarding protected regions on each of the nodes (SA_1, SA_2, SB_1, and SB_2). The task TA_2 on node A is activated by the arrival of a CAN message, which is queued by the task TB_1 on node B. Symmetrically, the task TB_2 on node B is activated by the arrival of a CAN message, which is queued by the task TA_1 on node A. The latter message has higher priority than the former, but both have higher priority than the low priority CAN messages that also are sent on the bus. In the table below, the deadline of tasks TA_2 and TB_2 are given relative the release of TB_1 and TA_1, respectively.

Task	Period T	Deadline D	Jitter J	Computation time C	Semaphores SA_1	SA_2	SB_1	SB_2
TA_1	100	-	3	7	2	1	-	-
TA_2	-	100	-	5	3	3	-	-
TA_3	50	25	2	10	4	2	-	-
TB_1	80	-	2	6	-	-	3	2
TB_2	-	100	-	5	-	-	2	1
TB_3	50	25	2	10	-	-	4	5

What we get here is three sets of coupled equations: one for the CAN bus and one for each of the nodes.

The basic equations for the nodes are on the form

$$R_i = J_i + w_i \tag{10.6}$$

where w_i is given by:

$$w_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \tag{10.7}$$

The equations for the CAN bus is on the form

$$R_i = J_i + w_i + C_i \tag{10.8}$$

where w_i is given by:

$$w_i = B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j + \tau_{bit}}{T_j} \right\rceil C_j \tag{10.9}$$

We will now instantiate these equations for the example above. This will reveal the true beauty of holistic scheduling!

The first things to decide are the priority orderings on both nodes. From Section 5.2 we know that $D - r^{max}$ gives an optimal priority ordering. Unfortunately we do not know D and r^{max} for all tasks. So what we will have to do for the time being is to define equations with some uninstantiated variables (and hope that we will be able to instantiate them later).

So how do we resolve the unknowns? Let's start looking at the deadline D_{TA_1}. TA_1 terminates by sending a CAN message to the node 2. This message will invoke task TB_2 which has a deadline of 100 relative the involation of TA_1. Consequently, D_{TA_1} can be defined as follows:

$$D_{TA_1} = 100 - R_{TB_2} - R_{CAN_1} \tag{10.10}$$

where R_{CAN_1} is the worst case response time for the CAN message sent from TA_1 to TB_2.

Similarly,

$$D_{TB_1} = 100 - R_{TA_2} - R_{CAN_2} \tag{10.11}$$

where R_{CAN_2} is the worst case response time for the CAN message sent from TB_1 to TA_2.

Now we know the deadlines of all tasks. What about the unknown task jitter (which will be equal to r^{max} since we assume that $r^{min} = 0$ for all tasks)?

Let's start with J_{TA_2}, which is dependendent on variations in the arrival of CAN_2 messages, i.e.

$$J_{TA_2} = R_{CAN_2} - 47\tau_{bit} \tag{10.12}$$

where $47\tau_{bit}$ is the minimum and R_{CAN_2} the maximum message response time (relative the release of TB_1).

Similarly,

$$J_{TB_2} = R_{CAN_1} - 47\tau_{bit} \tag{10.13}$$

So, now we have (at least symbolically) defined the priority orderings on the two nodes. Based on this we can calculate the blocking times (or at least define equations for the blocking times) using equation 4.5.

The only remaining unknowns in the table above are the periods for tasks TA_2 and TB_2. They are both kicked off by tasks (TB_1 and TA_1) with known periods, so they will of course inherit the periods of these tasks, i.e. $T_{TA_2} = 80$ and $T_{TB_2} = 100$. That was easy!

Now we have enough ammunition for defining the schedulability equations. Let's start with node 1:

$$
\begin{aligned}
R_{TA_1} &= 3 + w_{TA_1} && (10.14)\\
w_{TA_1} &= 7 + B_{TA_1} + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j && (10.15)\\
B_{TA_n} &= \max_{\{k,s \,|\, k \in lp(TA_n) \,\wedge\, s \in used\_by(k) \,\wedge\, ceil(s) \geq pri(TA_n)\}} cs_{k,s} && (10.16)\\
J_{TA_2} &= R_{CAN_2} - 47\tau_{bit} && (10.17)\\
R_{TA_2} &= J_{TA_2} + w_{TA_2} && (10.18)\\
w_{TA_2} &= 5 + B_{TA_2} + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j && (10.19)\\
R_{TA_3} &= 2 + w_{TA_3} && (10.20)\\
w_{TA_2} &= 10 + B_{TA_3} + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j && (10.21)
\end{aligned}
$$

The equations for the tasks on node 2 are symmetrical:

$$
\begin{aligned}
R_{TB_1} &= 2 + w_{TB_1} && (10.22)\\
w_{TB_1} &= 6 + B_{TB_1} + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j && (10.23)\\
B_{TB_n} &= \max_{\{k,s \,|\, k \in lp(TB_n) \,\wedge\, s \in used\_by(k) \,\wedge\, ceil(s) \geq pri(TB_n)\}} cs_{k,s} && (10.24)\\
J_{TB_2} &= R_{CAN_1} - 47\tau_{bit} && (10.25)\\
R_{TB_2} &= J_{TB_2} + w_{TB_2} && (10.26)\\
w_{TB_2} &= 5 + B_{TB_2} + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j && (10.27)\\
R_{TB_3} &= 2 + w_{TB_3} && (10.28)
\end{aligned}
$$

$$w_{TB_2} = 10 + B_{TB_3} + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \tag{10.29}$$

Remains to formulate the equations for the CAN-bus:

$$
\begin{aligned}
R_{CAN_1} &= R_{TA_1} + w_{CAN_1} + 135\tau_{bit} && (10.30)\\
w_{CAN_1} &= 135\tau_{bit} && (10.31)\\
R_{CAN_2} &= R_{TB_1} + w_{CAN_2} + 135\tau_{bit} && (10.32)\\
w_{CAN_1} &= 135\tau_{bit} + \left\lceil \frac{w_{CAN_1} + R_{TA_1}}{100} \right\rceil 135\tau_{bit} && (10.33)\\
& && (10.34)
\end{aligned}
$$

So this is it! We have a set of mutually dependent equations that can be solved with the iteration technique outlined in the beginning of this section. That is, we start by setting all variables (R_{TA_1}, R_{TA_2}, R_{TA_3}, R_{TB_1}, R_{TB_2}, R_{TB_3}, R_{CAN_1}, and R_{CAN_2}) to zero and then iteratively plug these values into the equation to obtain new variable values. The iteration stops either when all new values are equal to the previous values, or if any of the deadlines are violated.

You might think this is the end of the story. Wrong! Remember that we have priorities that are depending on deadlines, and that some of these deadlines are dependent on response times of some tasks and CAN communications. Consequently, the priority ordering might change when the values we are iterating change. This will affect the equations, leading to new response time values in the next iteration. This might change the priority ordering, ...

The result of all this is that the iterations might not terminate. Fortunately, the iterations terminate if we decide on a fixed priority ordering. Finding an optimal priority ordering is however not allways possible (since the problem is NP-hard). From a practical point of view this is not as bad as it might sound, since for most realistic task sets we will find a schedulable priority ordering (though not necessarily optimal) quite fast.

10.5 Summary

We have described the CAN bus protocol, showed how the timing analysis we had for processor scheduling could be applied to message scheduling, and showed how release jitter is inherited. The processor overheads due to handling messages was bounded. We showed how the inheritance of release jitter leads to the holistic scheduling problem. This problem can be overcome by iteration over successive approximations.

Chapter 11

Extensions to Fixed Priority Scheduling

11.1 Introduction

In this chapter we will look at how easy it is to extend the fixed priority scheduling analysis to handle a more complex model. It is important to be able to extend the model and the analysis so that the scheduling algorithm can be used in scheduling real applications.

11.2 Sporadically Periodic Tasks

Very often in a control system, we want to execute a task like this: at most every time T the task will be invoked. The task will run, be next invoked time t later, run again, be invoked t after that, and so on. There may be several invokations of the task in a *burst*. In effect, the task executes periodically with a period t when in a burst, and periodically (or sporadically) with a period no shorter than T between bursts.

We call the short period (the time between invokations within a burst) the *inner period*, and denote it t (so t_i is the inner period of a task i). We call the big period (the time between bursts) the *outer period*, and denote it T.

Why do we want to execute a task like this? Well, in many control systems we need a regular input from a sensor, at a period T. But the sensors are usually unreliable, and must be sampled several times with a small delay in between samples (so that we can average samples and reject bad readings). With the sporadically periodic model, we can set t to this small delay.

In order to analyse this, we are missing just one parameter: the maximum number of invokations in a burst. We denote this n (so n_i is the number of invokations of task i in a burst).

Clearly, the analysis we develop should be equivalent to our existing analysis when $T = t$ and $n = 1$ for all tasks. Just as a reminder, take another look at our existing analysis:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{11.1}$$

(B_i is the blocking factor of task i, and takes account of the time that a lower priority task can block task i: we looked at how to compute B_i in the previous chapter).

What needs to be changed with this analysis to include sporadically periodic tasks? Well, we might make no changes at all, and say that a sporadically periodic task j is invoked with time t_j between all invokations for ever more. That would be fine – the analysis would still be *sufficient* – but it would be over-estimating the effect of that task on lower priority tasks. If, for example, the outer period of task j is infinite (*ie.* the burst is invoked only once) then there can be at most n_j invokations of the task. But the analysis above would say that there are a lot more invokations of task j delaying task i – especially if R_i is large.

So what we need to do to update the above equation to handle sporadically periodic tasks better is to change the 'interference' term:

$$\left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{11.2}$$

This gives the maximum time that task j can delay task i whilst task i is executing. This delay we sometimes call the *interference*. In fact, in the above equation the term is made up from the maximum number of invokations multiplied by the maximum time an invokation can take. The term R_i in the equation is the longest time that task i can be running for – the worst-case response time of task i – but it can also be viewed as a 'window' over which we are measuring the interference from a higher priority task (later we'll see how this window isn't always the same as the worst-case response time, but for now we'll say that it's of duration R_i).

When we have 'bursty' behaviour, this interference term needs to be modified. We can do that by finding a better limit on the number of invokations whilst task i is running.

We need to make a restriction on the bursty behaviour though: we must have the burst taking no longer than the outer period (otherwise we don't have bursty behaviour!). So we must have $n_j t_j \leq T_j$.

The first question we ask is "How many complete bursts of a task j can fall in the window of duration R_i?" We can then multiply this number by n_j to give the number of invokations of task j in full bursts.

If we assume that the worst-case interference from task j happens when the start of a burst occurs at the same time that task i is released (the start of the 'window' of duration R_i) then there can be at most one burst that doesn't fall completely in the window: it can 'straddle' the end. A few invokations from the last burst could occur before the end of the 'window' (preempting task i, and hence delaying it). And a few would then occur after the end of the 'window' (too late to preempt task i, because it would have finished).

We should be able to see that the number of full bursts of task j is bounded by:

$$\left\lfloor \frac{R_i}{T_j} \right\rfloor \tag{11.3}$$

The function $\lfloor x \rfloor$ is the *floor* function, that rounds down to the nearest integer (it's the opposite of the *ceiling* function we're familiar with).

That's answered our first question. What about the second: how many invokations from a burst partially falling in the window?

We can answer that by knowing how long the window is, and how much of that window has been used by full bursts. There is time T_j between bursts of task j, so if we know the number of full bursts and multiply it by this time then we have the time taken up by full bursts in the window:

$$\left\lfloor \frac{R_i}{T_j} \right\rfloor T_j \tag{11.4}$$

The remaining time is the part of the window that a partial burst can execute over:

$$R_i - \left\lfloor \frac{R_i}{T_j} \right\rfloor T_j \tag{11.5}$$

This remaining time is, in effect, another window, over which task j will execute periodically with a period of t_j. We already know a bound for the number of invokations of tasks behaving like this: our original analysis contains the bound. But this time the window isn't of duration R_i. So the bound is:

$$\left\lceil \frac{R_i - \left\lfloor \frac{R_i}{T_j} \right\rfloor T_j}{t_j} \right\rceil \tag{11.6}$$

Now we can add this bound on the number of invokations in an incomplete burst to the number of invokations in complete bursts:

$$\left\lceil \frac{R_i - F_j T_j}{t_j} \right\rceil + n_j F_j \tag{11.7}$$

Where F_j is given by:

$$F_j = \left\lfloor \frac{R_i}{T_j} \right\rfloor \tag{11.8}$$

(we did this to simplify the equations a little bit).

Finally, we can put this back into our original analysis, replacing the term $\left\lceil \frac{R_i}{T_j} \right\rceil$:

$$R_i \;=\; C_i + B_i + \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i - F_j T_j}{t_j} \right\rceil + n_j F_j \right) C_j \tag{11.9}$$

And there we have it: new analysis taking account of bursty behaviour. Of course, if a given task j doesn't behave in a bursty manner, then we just set $n_j = 1$, and $t_j = T_j$.

The reader might like to prove that doing this for all tasks reduces the equations back to our original analysis (Hint: $\lceil x + a \rceil = \lceil x \rceil + a$ if a is an integer).

Note that Equation 11.9 can be extended. Because the final incomplete burst can consist of at most n_j invokations of j, the bound on the number can be changed from:

$$\left\lceil \frac{R_i - F_j T_j}{t_j} \right\rceil \tag{11.10}$$

to:

$$min\left(\left\lceil \frac{R_i - F_j T_j}{t_j} \right\rceil, n_j\right) \tag{11.11}$$

giving a final equation of:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left(min\left(\left\lceil \frac{R_i - F_j T_j}{t_j} \right\rceil, n_j\right) + n_j F_j \right) C_j \tag{11.12}$$

You should still be able to prove that if $n_j = 1$ and $T_j = t_j$ for all tasks then the updated equation for R will be the same as the original equation for R. To do this you will have to know $\lceil x \rceil - 1 = \lfloor x \rfloor$ iff x is not an integer.

11.3 Arbitrary Deadlines

Up until now we have assumed that deadlines of tasks are less than or equal to their periods ($D \leq T$). This is not always the case, so we need to extend our analysis to cope with this situation. The problem here is of course that a task now might interfere with itself, or more precisely: a previous invokation of a task may interfere with the current one. In the general case we might have a long history of interfering invokations.

Let's, for simplicity of the presentation, return to the simple model from Chapter 4 (ignoring jitter and sporadically periodic tasks). For the case when $D_i \leq T_i$ we then have the equation:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{11.13}$$

Assuming that tasks with identical priority are handled strictly in first come first served order (i.e. they are not allowed to preempt each other), this equation also holds for the first invokation even if we allow $D > T$. Let's use $R_i^{(1)}$ to denote the response time for the first invokation.

For the second invokation we have to extend the equation to handle the case when the first invokation interferes. This occurs when $R_i^{(1)} > T_i$. We are then in a situation with two successive invokations that execute without any idle time in between. Consequently, the behaviour in terms of worst case respons time for the second invokation is identical to that of a single invakation of a task with execution time $2C_i$. For that task, $i2$, we obtain the equation:

$$R_{i2} = 2C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_{i2}}{T_j} \right\rceil C_j \tag{11.14}$$

Since the response time R_{i2} is measured from the first invokation of task i (and since we know that the second invokation occurs T_i time units later) we obtain the following equation for the second invokation of task i:

$$R_i^{(2)} = R_{i2} - T_i \tag{11.15}$$

Assuming $R_i^{(2)} > T_i$ we obtain the following equation for the third invokation:

$$R_i^{(3)} = R_{i3} - 2T_i \tag{11.16}$$

where

$$R_{i3}^{(1)} = 3C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_{i3}}{T_j} \right\rceil C_j \tag{11.17}$$

and in the general case

$$R_i^{(n)} = R_{in} - (n-1)T_i \tag{11.18}$$

where

$$R_{in}^{(1)} = nC_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_{in}}{T_j} \right\rceil C_j \tag{11.19}$$

So, we have a method to calculate the worst case response time for the nth invokation of task i. But, what we are looking for is really a method to calculate the worst case response time after an arbitrary number of invokations. Fortunately, we know that if the utilisation $U \leq 1$ then there is always some point in the future when all invoked tasks have completed their execution. At that point, we know that the most recently invoked task has a response time which is less than its period. Consequently, the next task invokation will not be delayed by previous invokations. Hence, its worst case behaviour is identical to that of the first invokation.

This suggests a method for calculating R_i:

$k := 1$
`While` $R_i^{(k)} > T_i$ `do`
  Calculate $R_i^{(k)}$
  $k := k + 1$
`od` $R_i := \max_{l=1..k} R_i^{(l)}$

Chapter 12

The “Marble Sorter”

Chapter 13

Summary ...

13.1 Summary

13.1.1 Summary

Summary

Chapter 14

Problems

1. Below is a table which shows a task set of six tasks:

Task Name	T	D	C
A	1000	20	3
B	100	100	10
C	50	50	20
D	57	10	5
E	33	33	1
F	7	7	1

   (a) What is the utilisation of this task set?

   (b) What is the priority ordering if *rate monotonic* is used?

   (c) What is the priority ordering if *deadline monotonic* is used?

   (d) What are the worst-case response times of the task set in deadline monotonic priority order?

   (e) Are all deadlines met?

   (f) What are the worst-case response times of the task set in rate monotonic priority order?

   (g) Are all the deadlines met?

   (h) Add a task called FT with a period of 30, a deadline of 5, and worst-case computation time of 2. The new task set now looks like this:

Task Name	T	D	C
A	1000	20	3
B	100	100	10
C	50	50	20
D	57	10	5
E	33	33	1
F	7	7	1
FT	30	5	2

In deadline monotonic priority order, what are the new worst-case response times? Are all deadlines still met?

2. Below is a table which shows a task set of seven tasks:

Task Name	T	D	C
A	1000	20	3
B	100	100	10
C	50	50	20
D	57	10	5
E	33	33	1
F	7	7	1
FT	30	5	2

There are also four semaphores: S_1, S_2, S_3, and S_4. They are locked as follows:

Task Name	Semaphore Name	Held For
A	S_1	2
A	S_3	2
B	S_2	7
B	S_3	5
B	S_4	2
D	S_1	2
C	S_2	1
FT	S_1	1

(a) Assuming the deadline monotonic priority ordering, what are the priority ceilings of the four semaphores?

(b) What are the blocking factors of the seven tasks?

(c) What are the worst-case response times of the tasks with these blocking factors?

(d) Are all deadlines met?

3. Below is a table which shows a task set of six tasks:

Task Name	T	D	C
A	35	35	9
B	7	7	2
C	60	50	5
D	1000	30	10
E	30	20	3
F	60	55	10

There are also two semaphores: S_1 and S_2. They are locked as follows:

Task Name	Semaphore Name	Held For
A	S_1	2
C	S_2	2
E	S_1	3
F	S_2	5

(a) What are the worst-case response times of these tasks if deadline monotonic ordering is used?

(b) Are all deadlines met?

(c) Now modify task B so that instead of executing always with a period, it has an inner period (t) of 7, and outer period (T) of 75, and executes three times ($n = 3$) in each "burst". The table gives the modified task set:

Task Name	T	t	n	D	C
A	35	35	1	35	9
B	75	7	3	7	2
C	60	60	1	50	5
D	1000	1000	1	30	10
E	30	30	1	20	3
F	60	60	1	55	10

Using the extended analysis and the same priority order, what are the worst-case response times of the tasks?

(d) Are all deadlines met?

(e) The task set is modified to include release jitter:

Task Name	T	t	n	D	C	J
A	35	35	1	35	9	0
B	75	7	3	7	2	0
C	60	60	1	50	5	0
D	1000	1000	1	30	10	0
E	30	30	1	20	3	14
F	60	60	1	55	10	0

Modify the analysis for bursty behaviour to also include release jitter (make sure that the worst-case response time includes the delay before release).

(f) What are the worst-case response times of all the tasks with the deadline monotonic priority ordering?

(g) Is the deadline of task E met? Are other deadlines missed?

(h) What are the worst-case response times of all the tasks with the following priority ordering: E, B, D, A, C, F.

(i) Is the deadline of task E met? Are other deadlines missed? What is the optimal priority algorithm?

4. Below is a table which shows a task set of four tasks:

Task Name	T	D	C
A	70	70	7
B	50	50	1
C	60	60	2
D	1000	30	8

(a) Assuming that a tick driven scheduler is used, with the following overheads: The context switch cost C_{sw} is 1. The cost of moving a task from the delay queue to the run queue, C_{queue} is 2. The cost of responding to the timer interrupt, C_{tick}, is 1. The tick period, T_{tick}, is 7.

What are the worst-case response times of these tasks in the optimal priority order?

(b) Are all deadlines met?

(c) The tick period T_{tick} now changes to 13. What are the new worst-case response times?

(d) The scheduler is now changed to an event scheduler. The cost of responding to a timer interrupt and of moving a task from the delay queue to the run queue, C_{timer}, is 3. What are the worst-case response times of tasks now?

(e) Are all deadlines met?

5. Below is a table which shows a task set of seven messages to be sent on a CAN bus:

Message Name	T	D	s
A	50	5	3
B	5	5	2
C	10	10	1
D	50	20	1
E	50	20	5
F	100	100	6
G	1000	1000	1

The times (T and D) given are in milliseconds, and the sizes (s) are given in bytes. The bus speed is 50 Kbits/sec. The queuing jitter for all messages is zero.

The tasks are listed in priority order (this is the deadline monotonic priority ordering).

(a) What is τ_{bit} for this bus speed?

(b) What are the transmission times (C) of the messages?

(c) What are the worst-case response times (R) for each message? (you may assume a blocking factor for all messages equivalent to the transmission time of an 8 byte message).

(d) What is the bus utilisation?

(e) Are all deadlines met?

Chapter 15

Answers to Problems

1(a) The utilisation of the task set is $\approx 76.4\%$.

$$U = \sum_{i=1}^{i=n} \left(\frac{C_i}{T_i}\right) \approx 0.764$$

1(b) The *rate monotonic* priority order is **F, E, C, D, B, A** (from highest to lowest priority).

1(c) The *deadline monotonic* priority order is **F, D, A, E, C, B** (from highest to lowest priority).

1(d) The worst-case response times of the task set in deadline monotonic priority order are $\mathbf{R_A = 10, R_B = 47, R_C = 35, R_D = 6, R_E = 11, R_F = 1}$.

1(e) Yes, all deadlines are met.

1(f) The worst-case response times of the task set in rate monotonic priority order are $\mathbf{R_A = 47, R_B = 44, R_C = 25, R_D = 31, R_E = 2, R_F = 1}$.

1(g) No, all deadlines are not met (task A and D).

1(h) The new worst-case response times, with an added task FT and the tasks in deadline monotonic priority order are
$\mathbf{R_A = 12, R_B = 84, R_C = 40, R_D = 9, R_E = 13, R_F = 3, R_{FT} = 2}$.
All the deadlines are still met.

2(a) Assuming deadline monotonic priority ordering, the ceilings of the four semaphores are $\mathbf{S_1 = 1, S_2 = 6, S_3 = 4, S_4 = 7}$.

2(b) and **2(c)** The blocking factors and the worst-case response times in deadline monotonic priority order are given in the table below.

Task Name	T	D	C	**B**	**R**
A	1000	20	3	**5**	**18**
B	100	100	10	**0**	**84**
C	50	50	20	**7**	**48**
D	57	10	5	**2**	**11**
E	33	33	1	**5**	**19**
F	7	7	1	**2**	**5**
FT	30	5	2	**2**	**4**

2(d) No, all deadlines are not met.

3(a) The worst-case response times of the tasks, if deadline monotonic ordering is used, are
$\mathbf{R_A = 35, R_B = 2, R_C = 67^{\dagger}, R_D = 21, R_E = 7, R_F = 97^{\dagger}}$.
† The response time went over the period time for the task.

3(b) No, the deadlines are not met.

3(c) The worst-case response times of the tasks are
$\mathbf{R_A = 28, R_B = 2, R_C = 50, R_D = 21, R_E = 7, R_F = 55}$.

3(d) Yes, the deadlines are met.

3(e) Analysis for bursty behaviour including release jitter:

$$w_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left(min \left(\left\lceil \frac{w_i + J_j - F_j T_j}{t_j} \right\rceil, n_j \right) + n_j F_j \right) C_j$$

$$F_j = \left\lfloor \frac{w_i + J_j}{T_j} \right\rfloor \qquad R_i = J_i + w_i$$

3(f) The worst-case response times with deadline monotonic ordering are $\mathbf{R_A = 31, R_B = 2, R_C = 53, R_D = 24, R_E = 21, R_F = 58}$.

3(g) The deadline of task E is **not** met. The tasks C and F also miss their deadlines.

3(h) The worst-case response times with the priority order E, B, D, A, C, F are $\mathbf{R_A = 31, R_B = 7, R_C = 53, R_D = 24, R_E = 19, R_F = 58}$.

3(i) The deadline of task E is met, but the deadlines of task C and F are missed. The optimal priority algorithm is to set the priorities in order $D - J$ instead of D (deadline monotonic) order.

4(a) The worst-case response times in optimal priority order, i.e. *deadline monotonic* order, are $\mathbf{R_A = 47, R_B = 32, R_C = 37, R_D = 28}$.

4(b) Yes, all the deadlines are met.

4(c) With the tick period $T_{tick} = 13$ the new worst-case response times are $\mathbf{R_A = 50, R_B = 36, R_C = 41, R_D = 33}$.

4(d) The scheduler is now changed to an event scheduler. The cost of responding to a timer interrupt is $C_{timer} = 3$. This gives the worst-case response times $\mathbf{R_A = 38, R_B = 25, R_C = 29, R_D = 22}$.

4(e) Yes, all the deadlines are met.

5(a) The bus speed $50Kbits/sec$ gives $\tau_{bit} = 20\mu s$.

5(b) The transmission times of the messages given in milliseconds are
$\mathbf{C_A = 1.64, C_B = 1.46, C_C = 1.26, C_D = 1.26, C_E = 2.02,}$
$\mathbf{C_F = 2.22, C_G = 1.26}$.

5(c) Here we assume that the blocking factor for all messages are equivalent to the transmission time of an 8-byte message i.e. $135\tau_{bit}$. When deadline monotonic priority order is used the messages A and B have the same priority as well as message D and E. Because of that we have to calculate the worst-case response times for the different combinations of these messages. We will use the following priority orders:
$\mathbf{P1 : A, B, C, D, E, F, G}$
$\mathbf{P2 : B, A, C, E, D, F, G}$
The results are given in the table below. (All times given in milliseconds.)

† The response time went over the period time for the task.

If we for instance only change the order of message A and B (i.e. priority order B, A, C, D, E, F, G) then the worst-case response times of the lower priority messages C to G will stay unaffected (column R_{P1}, rows C to G in the table above). Message A and B will have the response times found in column R_{P2}.

5(d) The utilisation is $\approx 54\%$.

5(e) No, all deadlines are not met.

Message Name	T	D	s	C	$\mathbf{R_{P1}}$	$\mathbf{R_{P2}}$
A	50	5	3	1.64	**4.24**	**5.70**
B	5	5	2	1.46	**5.70**†	**4.06**
C	10	10	1	1.26	**8.42**	**8.42**
D	50	20	1	1.26	**9.68**	**14.42**
E	50	20	5	2.02	**11.70**	**10.44**
F	100	100	6	2.22	**16.64**	**16.64**
G	1000	1000	1	1.26	**19.36**	**19.36**