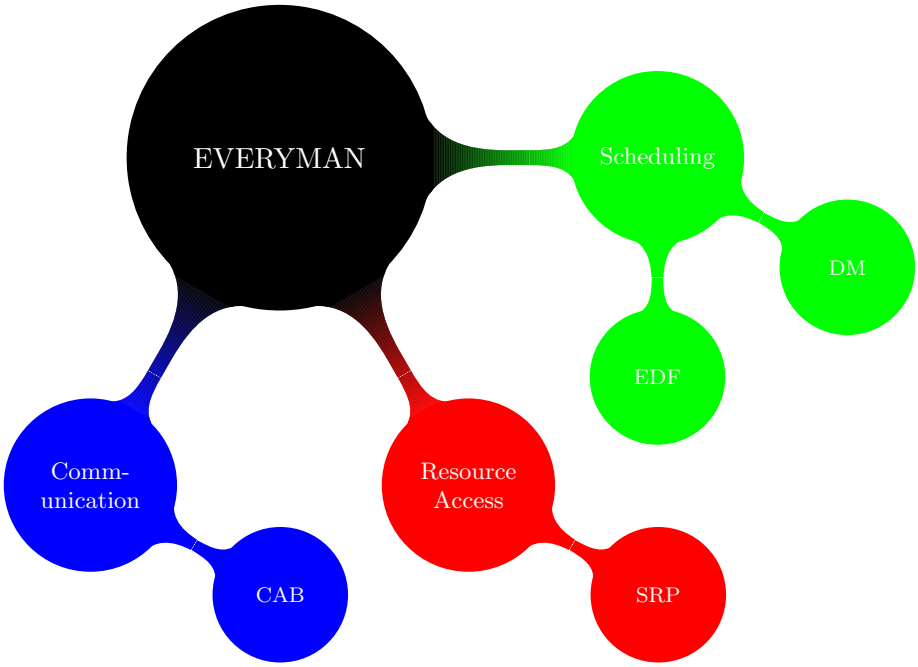
The Design of the Everyman Hard Real-time Kernel

Barry Watson



CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

INTRODUCTION

The goal of this book is to introduce the EVERYMAN hard real-time kernel. The kernel has been designed to be suitable for hard real-time systems development, to be easily understood by a single programmer, and to use as little memory as possible.

The book is divided into three steps.

Step 1 Here we'll see two scheduling algorithms for hard-real time systems. We then go on to the design and implementation of a kernel which implements these algorithms.

Step 2 In this step we'll take a look at resource access protocols used with semaphores. We'll then build upon the kernel described in Step 1 and implement semaphores which eliminate deadlock, livelock, and unbounded blocking.

Step 3 We then move on to discuss the communication primitives used in modern operating systems and their applicability to hard real-time systems. We'll then build upon the kernel described in Step 2 and implement a communication primitive which is suited for the needs of hard real-time systems.

At the end of the book you will have seen the design and implementation of the EVERYMAN kernel and hopefully have a better understanding of what it takes to build reliable hard real-time systems. You should also have the required knowledge to evaluate the kernels and operating systems that are available in today's market.

STEP 1

In this step build a basic kernel which schedules independent jobs depending upon their timing characteristics.

1.1 What is hard real-time?

Sometimes we build computer systems that are not self contained. These systems have to interact either with the physical world we live in or other computer systems. It is often the case that these external environments place timing constraints on the systems we build. To give some examples, I've been involved with the construction of software that had to sample the temperature of molten iron at a fixed frequency so as to accurately determine graphite content via the shape of a cooling curve. I've also helped write software that controlled a slave microprocessor that had to react to a serial bus clock signal from its master within a certain deadline to avoid a crash. These monitoring and control applications are textbook examples of real-time systems — systems which consist of a set of code routines each of which need to execute at a specified frequency and are constrained to finish their execution within a certain deadline. Depending on just how "hard" this constraint is we can divide systems into two groups. Not surprisingly these groups are called soft and hard. A soft real-time system can tolerate a missed deadline but system performance is degraded. A hard real-time system on the other hand will see a missed deadline as a complete failure. The temperature measuring example was soft; if our timing was off it didn't matter that much as the metallurgist's calculation algorithms could tolerate sampling jitter. The serial bus clock example I gave was hard; if we missed the timing then the system would crash; we had no workaround.

In this book we'll focus on hard real-time systems. In some ways, hard real-time is simpler than soft real-time.

1.2 The need for scheduling

If we build reasonably complex real-time systems then we'll have several different code routines each with their own periods of execution. Sooner or later we'll have a situation where we have more code routines that need to be executed than we have processors that can execute them. To solve this problem we need some kind of mechanism that decides which code routine will receive processor attention. This mechanism is called a scheduling algorithm and the bright minds of industry and academia have come up with such algorithms for real-time systems along with mathematical tools which give us implementors the chance to analyse our systems to ensure we can meet our deadlines.

We're going to look at two of these scheduling algorithms but before we get into the specifics of algorithms and mathematics, we need to agree on some terms. All of the algorithms you will see traffic in jobs. These are the basic units of scheduling and we'll denote them with the symbol J. Each job will be assigned a code function to be executed by a processor. Since the scheduler doesn't need to know the specifics of the function we won't give it a symbol here. However, the scheduler does need to know how much processor time this code function will need to execute in the worst case. This time is denoted with

the symbol C. In figure 1.1 we see a job which takes $C = 2$ time units to execute.

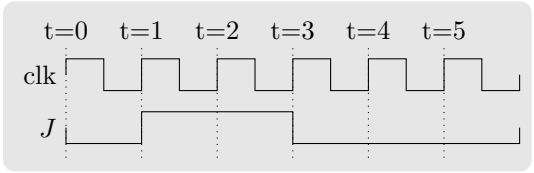


Figure 1.1: The job.

Because the code is to be executed at a fixed frequency, we'll need to give each job a period which will determine the time between separate instances of code execution. The period is denoted by T and we'll assume that all jobs start their first period at time zero. As real-time code executions have a deadline constraint, we'll also give each job a deadline relative to the the start of each job execution period. This is denoted by D. Figure 1.2 shows these timing specifications graphically.

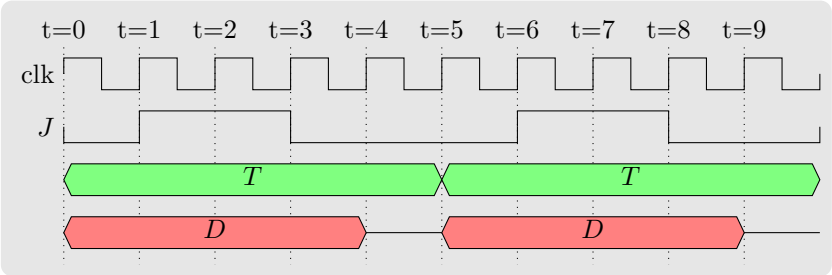


Figure 1.2: Job Timing.

As far as any scheduling algorithm is concerned, jobs will be in one of three states: current, pending, or, inactive. The current state will be used for a job which is currently executing. We're only going to discuss uniprocessor machines so at any given time no more than one job can be in the current state. Note that it may be possible for no jobs to be in the current state. An inactive job is one which is waiting for the the start of its next period of activation. These inactive jobs do not compete for the processor. When the time comes for the start of the next period for an inactive job then that job will move into the pending state. All pending jobs compete for the processor's time. As we can see, jobs will move from inactive to pending, then from pending to current. When they have finished executing they will be inactive again.

Moving on to mathematics, we'll need to understand two functions. The floor function, which is written like this $\lfloor x \rfloor$, calculates the largest integer that is less than or equal to x. For example $\lfloor 3.14 \rfloor = 3$, $\lfloor -3.14 \rfloor = -4$, and $\lfloor 3 \rfloor = 3$. The ceiling function, which is written like this $\lceil x \rceil$, calculates the smallest integer that is greater than or equal to x. For example $\lceil 3.14 \rceil = 4$, $\lceil -3.14 \rceil = -3$, and $\lceil 3 \rceil = 3$. These two functions help us reason in terms of job

periods. Let's say we have a job with a period of 3 time units. If the system runs for 10 time units then we know that the job has $\frac{10}{3} = 3.333$ periods. $\lceil \frac{10}{3} \rceil = 4$ tells us how many times the job's period has started. $\lfloor \frac{10}{3} \rfloor = 3$ tells us how many complete periods have occurred.

1.3 The theory behind real-time scheduling

Now we're ready to look at two real-time scheduling algorithms. These algorithms will be priority driven and preemptive. Every job in the current or pending state will have a priority associated with it. These algorithms are designed to make sure that at any given time the job in the current state has a priority higher than, or at least equal to, the highest priority pending job. Our two algorithms only differ from each other in what priority they associate with a job.

As time progresses, it may well be the case that a job moves from the inactive state to the pending state and that this job actually has a higher priority than the job in the current state. If this should happen then the current job has its execution suspended and it is moved back into the pending state. The highest priority job among all pending jobs is then moved into the current state. This is called a preemption and it is shown graphically in figure 1.3 where J_2 is preempted at $t = 4$ by J_1 which has a higher priority.

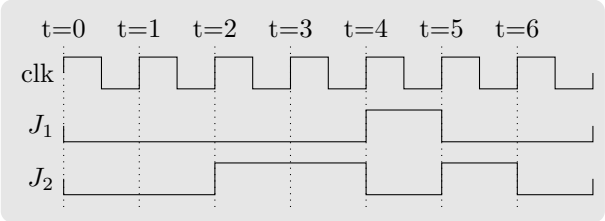


Figure 1.3: Job preemption.

Each algorithm that is discussed will also have a feasibility test. If our job sets pass the test then we'll have a guarantee that every job will always meet its deadline.

Deadline Monotonic

The year 1973 saw the introduction of the Rate Monotonic (RM) algorithm [CL73]. The priority assignment rule with this algorithm is very simple. A job with a certain frequency of activation is given a higher priority than any job with a lower frequency of activation. To describe this with terms we have already introduced, a job's priority is inversely proportional to its period.

With the assumption that a job's relative deadline was equal to the job's period, the inventors of the algorithm proved that a set of n jobs could be successfully scheduled with RM if the following condition held:

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

In this equation $\frac{C_i}{T_i}$ is the fraction of the total processor time taken up by J_i. The algorithm's inventors also showed that as n increases, $n(2^{\frac{1}{n}}-1)$ approaches 0.69. This equation is an example of a feasibility test. This particular test is very easy to apply: just sum up all the $\frac{C_i}{T_i}$ and compare the result with 0.69. However, the RM feasibility test is what is known as sufficient but not necessary. That is to say there may be a set of jobs which leads to a processor utilisation of over $n(2^{\frac{1}{n}}-1)$ which may still have a feasible schedule. Such jobs sets are characterised by harmonic job activation frequencies.

One of the problems with RM is that many systems will need job deadlines shorter than the job's period which violates the assumption mentioned earlier. A solution to this problem arrived in 1982 with the introduction of the Deadline Monotonic (DM) algorithm [JL82]. With DM, a job's priority is inversely proportional to its relative deadline. That is to say, the shorter the relative deadline, the higher the priority. RM can be seen as a special case of DM where each job's relative deadline is equal to the its period. However, the similarities end there. The "69%" feasibility test which we saw earlier doesn't work with DM.

The DM feasibility test will involve calculating how long it takes a job to go from the start of its period to the point where it finishes execution. We'll call this length of time the response time and denote it with R. After calculating R we then compare it with the job's relative deadline. If it is shorter, then this job passes the test, otherwise it fails because a deadline can be missed. We have to check the feasibility of every job we define.

Let's assume we have a set of n jobs, $\{J_1, ..., J_n\}$, where J_1 has the highest priority and J_n the lowest. We can say that for a given job J_i, R_i will be equal to the job's worst case execution time C_i plus all of the interference given by the execution of higher priority jobs. Our first guess at this interference time could be $\sum_{k=1}^{i-1} \left\lceil \frac{C_i}{T_k} \right\rceil C_k$. The term $\left\lceil \frac{C_i}{T_k} \right\rceil$ gives us the number of times J_k executes during C_i and when we multiply this with C_k we have the total interference from J_k. However, this isn't quite correct. The problem lies with the use of $\left\lceil \frac{C_i}{T_k} \right\rceil$. We need to calculate the total interference during the response time, not just during C_i which is only the processor time needed for J_i. What we want is $\left\lceil \frac{R_i}{T_k} \right\rceil$, which leads us to

$$R_i = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Since R_i appears on both sides of the equation we have a recurrence relation and we need to solve it for R_i. To do this we take a first "guess" at the response time and use this for the R_i on the right hand side then calculate the left hand side R_i. If the calculated R_i differs from the guessed R_i then we just substitute the left hand side R_i into the right hand side and recalculate. We repeat this until we calculate the same value for R_i twice. Unlike the RM feasibility test, this test is both sufficient and necessary. That is to say, job sets that pass the test are guaranteed to meet every deadline. Job sets that fail can experience missed deadlines.

Algorithms are generally best explained by showing examples. Table 1.1 shows an example set of jobs with period, execution time, and deadline. We'll

Table 1.1: Deadline Monotonic example job set.

Job	T	C	D
J_1	10	1	5
J_2	15	3	10
J_3	100	50	75

now calculate the response time for J_3 and see if it is within the deadline of 75. First we take the initial guess at R_3 to be equal to C_3, that is to say 50.

$$R_3 = 50 + \sum_{k=1}^{2} \left\lceil \frac{50}{T_k} \right\rceil C_k$$

$$R_3 = 50 + \left\lceil \frac{50}{10} \right\rceil 1 + \left\lceil \frac{50}{15} \right\rceil 3 = 50 + 5 + 12 = 67$$

Our calculated R_3 doesn't equal our guess of 50 so we substitute 67 for R_3 on the right hand side and recalculate.

$$R_3 = 50 + \left\lceil \frac{67}{10} \right\rceil 1 + \left\lceil \frac{67}{15} \right\rceil 3 = 50 + 7 + 15 = 72$$

Substitute again

$$R_3 = 50 + \left\lceil \frac{72}{10} \right\rceil 1 + \left\lceil \frac{72}{15} \right\rceil 3 = 50 + 8 + 15 = 73$$

And again

$$R_3 = 50 + \left\lceil \frac{73}{10} \right\rceil 1 + \left\lceil \frac{73}{15} \right\rceil 3 = 50 + 8 + 15 = 73$$

We see that R_3 has stabilised at 73 and this is the response time for J_3. Since 73 is less than the relative deadline for J_3 which is 75, we know that this job can be scheduled.

The relationships between job priorities assigned with DM are static. If J_k has a higher priority than J_l then this never changes. This is quite useful if you have a real-time kernel which implements a priority driven first come first served scheduling algorithm, e.g. POSIX.4 real-time extensions to Unix [Gal95]. In these cases we just think of our jobs being processes and assign our priorities according to DM then let the system's scheduler take care of the rest.

Earliest Deadline First

Our next algorithm is called Earliest Deadline First (EDF). With EDF, priority assignment is inversely proportional to absolute deadline. If we take two jobs, J_1 ($T_1 = 3$, $D_1 = 3$) and J_2 ($T_2 = 10$, $D_2 = 10$), we see that with DM, J_1 would always have a higher priority than J_2. However, with EDF, J_1 would have a higher priority than J_2 for its first three periods but not its fourth period where the absolute deadline for J_1 would be 12 which is greater than the absolute deadline for J_2 which is 10. The relationship between job priorities is, as you can see, dynamic. This unfortunately causes problems with the feasibility

test we used with DM. That test assumed job priorities were static when we calculated the interference from higher priority jobs. This means we cannot apply the feasibility test we defined for use with DM.

The basic idea behind the EDF feasibility test is to calculate how much processor time is needed at every absolute deadline in the schedule and check that this doesn't exceed the actual processor time available.

At first glance we might think that if our system runs for many years then it would take a long time to analyse every deadline. However, we don't need to consider the system's entire lifetime. Since we're dealing with periodic jobs, we find that our schedules are built up from patterns, or subschedules, which are repeated time and time again. Imagine two jobs with $T_1 = 3$ and $T_2 = 4$, looking at a schedule we would see the same pattern every 12 time units. With $T_1 = 5$ and $T_2 = 10$, we would see a pattern repeated every 10 time units. If we can guarantee the feasibility of one of these subschedules then we would have a guarantee for the lifetime of the system. The length of such a subschedule has a name, it's called a hyperperiod, and it is equal to the least common multiple (lcm) of all the periods in the job set. The lcm of two periods is defined as

$$\operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

The function gcd used in the definition is the classic greatest common divisor function which has probably been implemented by every student of programming.

$$\gcd(a, b) = \begin{cases} b & \text{if } a = 0 \\ \gcd(b \bmod a, a) & \text{otherwise} \end{cases}$$

To calculate the lcm of lots of periods we only have to know that $\operatorname{lcm}(a, b, c) = \operatorname{lcm}(a, \operatorname{lcm}(b, c))$.

If we let L be any time between zero and the hyperperiod, then we can say that the number of periods of a job J_i that have a deadline before or at L is equal to $\left\lfloor \frac{L-D_i}{T_i} \right\rfloor + 1$. To understand how this works take a look at figure 1.4 which shows a schedule where a job has two periods. This job has only one deadline before or at the time L. If we start at time L and move back D time units, $(L - D)$, then we'll end up somewhere in the first period and if we remember that the floor function gives us the number of fully completed periods (in this example zero) then $\left\lfloor \frac{L-D}{T} \right\rfloor + 1 = 1$ which is the result we were looking for. The point L' is after the second deadline so $L' - D$ will be in the second period, therefore $\left\lfloor \frac{L'-D}{T} \right\rfloor + 1 = 2$.

Let's define the processor demand at any time L to be the total execution time required by all jobs with deadlines before or at L. If we have a set of n jobs, we can write this mathematically as: $\sum_{i=1}^{n} \left(\left\lfloor \frac{L-D_i}{T_i} \right\rfloor + 1 \right) C_i$

Because $\left\lfloor \frac{L-D_i}{T_i} \right\rfloor$ only changes when L is a multiple of D_i we need only be concerned with checking the processor demand at every deadline. So, we check every deadline up to the hyperperiod of the job set and if for every deadline the sum of all of the execution times for every job is less than L then we know we have a feasible schedule. Let $\mathbb{D}$ be the set of all of the absolute deadlines

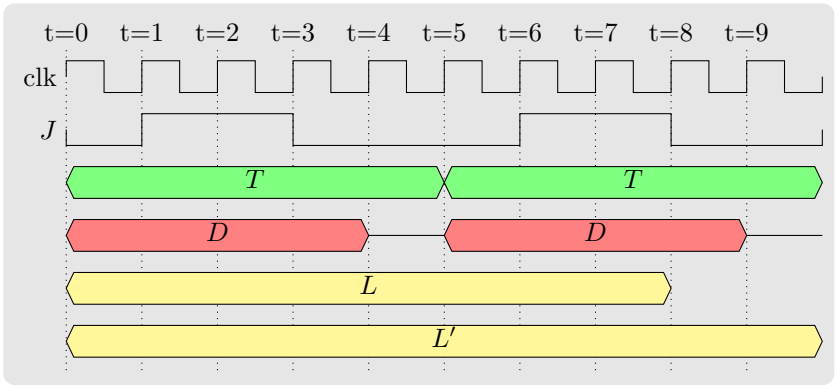


Figure 1.4: Processor demand.

for every job up to the hyperperiod. Our feasibility test will be

$$\forall L \in \mathbb{D}, L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

This test is both necessary and sufficient.

Table 1.2: Earliest Deadline First example job set.

Job	T	C	D
J_1	3	1	2
J_2	4	2	3
J_3	12	2	11

Again, it is easier to understand the theory when we have an example to look at. Table 1.2 describes the timing specifications of three jobs. The hyperperiod for these three jobs is 12. The absolute deadlines we're interested in for J_1 are at times 2, 5, 8, and 11. For J_2 we have absolute deadlines at times 3, 7, and 11. For J_3 we have only one absolute deadline at time 11. Therefore $\mathbb{D} = \{2, 3, 5, 7, 8, 11\}$. Let's take each member of this set in turn and calculate the feasibility of this job set with EDF.

First, we take $L = 2$

$$2 \geq \sum_{i=1}^{3} \left(\left\lfloor \frac{2 - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

$$2 \geq \left(\left\lfloor \frac{2-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{2-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{2-11}{12} \right\rfloor + 1 \right) 2$$

$$2 \geq 1 + 0 + 0$$

It holds, i.e. the job set is feasible at time 2.

We then take $L = 3$

$$3 \geq \left(\left\lfloor \frac{3-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{3-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{3-11}{12} \right\rfloor + 1 \right) 2$$

$$3 \geq 1 + 2 + 0$$

It holds, i.e. the job set is feasible at time 3.

We move on to $L = 5$

$$5 \geq \left(\left\lfloor \frac{5-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{5-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{5-11}{12} \right\rfloor + 1 \right) 2$$

$$5 \geq 2 + 2 + 0$$

It holds, i.e. the job set is feasible at time 5.

For our next member of $\mathbb{D}$ we have $L = 7$

$$7 \geq \left(\left\lfloor \frac{7-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{7-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{7-11}{12} \right\rfloor + 1 \right) 2$$

$$7 \geq 2 + 4 + 0$$

It holds, i.e. the job set is feasible at time 7.

We now take $L = 8$

$$8 \geq \left(\left\lfloor \frac{8-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{8-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{8-11}{12} \right\rfloor + 1 \right) 2$$

$$8 \geq 3 + 4 + 0$$

It holds, i.e. the job set is feasible at time 8.

And finally, we take $L = 11$

$$11 \geq \left(\left\lfloor \frac{11-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{11-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{11-11}{12} \right\rfloor + 1 \right) 2$$

$$11 \geq 4 + 6 + 2$$

It does not hold, i.e. the job set is not feasible. The deadline at time 11 for J_3 will be missed. Figure 1.5 shows the schedule of the above job set. We can see that J_3 hasn't finished execution at time 11. The extra unit of execution at time 11 is beyond that job's deadline. The job J_2 also has a deadline at time 11. This means J_2 and J_3 have the same priority at this point and according to EDF it could easily have been J_2 which misses its deadline instead of J_3. In any case this real-time system has failed.

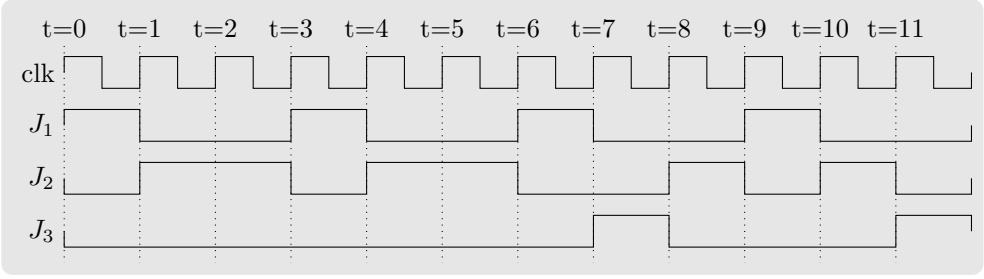


Figure 1.5: EDF schedule.

If all jobs have their relative deadlines equal to their period then there is a simpler feasibility test [CL73]. We can calculate what fraction of a job's period is taken up by its execution time, add up all the results for every job, then test to see the result this is less than 100%. This can be written mathematically as follows:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

If this condition holds then we know that EDF will generate a feasible schedule for our job set. If we take the example job set given in table 1.2 and increased each job's deadline to be equal to its relative deadline then we would have a processor utilisation factor of:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} = \frac{1}{3} + \frac{2}{4} + \frac{2}{12} = \frac{4}{12} + \frac{6}{12} + \frac{2}{12} = 1$$

This job set would in that case be feasible with EDF.

1.4 Hidden costs

The treatment given to scheduling so far has been quite theoretical. The algorithms we've seen are time driven; they need a hardware clock which generates interrupts at a constant frequency each of which could be the start of a job's period and a possible preemption. We haven't mentioned how to account for the time it takes the kernel to schedule jobs, nor have we mentioned how to account for interference from hardware interrupts. Apart from interrupts and scheduling costs, there are other hidden costs which can make the worst case execution time calculations a bit tricky.

Accounting for interrupts

Hardware interrupts can be problematic to deal with since the service routines effectively have higher priorities than any system job. That is to say the currently executing job has no option but to suspend execution whilst the interrupt service routine executes. Our scheduling tests do not take these service routines into consideration and this makes their results too optimistic and therefore unusable. Our only option is to try to "work in" the interrupt costs into our feasibility tests.

When we account for the interrupt service routine costs, we can model them like we did with jobs; each routine will have a worst case execution time (C) and a worst case interarrival time [1] (T). We won't give the routines a relative deadline as we're not trying to test their feasibility as they are scheduled by interrupt controllers not our algorithms.

If we have m interrupts then we can calculate that under a given duration of time, L, then the interference from all interrupts would be $\sum_{j=1}^{m} \left\lceil \frac{L}{T_j} \right\rceil C_j$. The amount of processor time which is needed to handle interrupts under a duration of time L is given by the following function:

[1] The minimum time between interrupts.

$$\mathrm{f}(0) = 0$$
$$\mathrm{f}(L) = \begin{cases} \mathrm{f}(L-1) + 1 & \text{if } \sum_{j=1}^{m} \left\lceil \frac{L}{T_j} \right\rceil C_j > \mathrm{f}(L-1) \\ \mathrm{f}(L-1) & \text{otherwise} \end{cases}$$

Table 1.3: Interrupt example.

Interrupt	T	C
Int_1	3	1
Int_2	6	2

Table 1.4: Interrupt handling cost and interference.

L	$\mathrm{f}(L)$	$\sum_{j=1}^{2} \left\lceil \frac{L}{T_j} \right\rceil C_j$
0	0	0
1	1	3
2	2	3
3	3	3
4	4	4
5	4	4
6	4	4

We can quickly show the difference between interrupt handling costs and interrupt interference with an example. Table 1.3 shows two interrupts, their worst case interrival time, and their worst case execution time. Table 1.4 shows the cost of handling these interrupts during the time from 0 to 6, and the interference experienced by jobs which have a lower priority during the same time. To "work in" interrupt handling we merely expand our feasibility tests to include either handling costs or interference.

In the case of EDF scheduling we would have a new feasibility test of:

$$\forall L \in \mathbb{D}, L - \mathrm{f}(L) \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

What we're saying here is that the processor time available at time L, minus all of the time required for interrupt handling, must be grater than or equal to the time required for job execution [KJ93].

For DM scheduling we would have to change the response time calculation in the feasibility test to include the interference experienced by interrupt processing. The new response time equation will then be:

$$R_i = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k + \sum_{j=1}^{m} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

As far as the hardware clock is concerned, its interrupts are generated at a constant frequency so the value of T is quite easy to figure out. The execution

time associated with the clock interrupt would be all the work that is common to every clock tick, e.g., updating the global system time, checking all inactive jobs to see if their period has started, and possibly checking every job in the current and pending states for a missed deadline. We don't associate the work required for scheduling and preemption since these costs are not incurred with every clock interrupt. We'll look at those costs a little later.

Other hardware interrupts may be more problematic especially if there is a lot of work to be done with each interrupt. Imagine we were writing code to control a network controller chip which generated interrupts, and that each interrupt involved the transport of data between memory and the network controller. This might be too much work for a code routine with a priority higher than any job. To avoid the problem of too much interrupt service execution time we could attempt to split the work performed at every interrupt into a small interrupt service routine which recognises the network interrupt, and a job which performs the actual processing and controlling of the hardware. In essence, instead of being a time triggered (periodic) job, we define the network job to be event triggered (sporadic). Like any other periodic job this sporadic network job would have a code function and deadline. The network controller interrupt when activated would place its sporadic job into the pending state and the scheduling algorithm would deal with it just like a periodic job. The only thing left to do is find the minimum amount of time [2] between network controller interrupts. This would be the network job's period. We would then have all the information we need to perform the feasibility test.

If we think about it, the only difference between periodic and sporadic jobs is the event that moves the job into the pending state — hardware clock interrupt vs. network controller interrupt. In many cases we may be tempted to just poll the network controller with a periodic job which had a suitable period. In general, periodic polling can be seen as a low-pass filter protecting against erroneous input overflow from the environment. This is a technique used exclusively in some hard real-time systems [SZ89].

Accounting for scheduling

Now we've dealt with interrupts we can look at the costs connected with scheduling. During its period, a job may be preempted many times but it preempts only once. For every job period we have these scheduling costs:

Period arrival The job's period arrives and it is moved from the inactive state into the pending state.

Preemption prologue The job eventually moves into the current state possibly preempting a lower priority job. Execution is then started.

Preemption epilogue Execution finishes and the job is moved into the inactive state. The job which was initially preempted is moved back into the current state.

If we extend every job's worst case execution time with these costs then we can integrate the cost of scheduling into our feasibility tests without altering the equations.

[2] The worst case interarrival time.

Other costs

When the bus between the processor and memory is shared with hardware devices which use Direct Memory Access (DMA) the worst case execution time C may be difficult to determine. A DMA controller often works on the basis of cycle stealing. When the controller needs to access the bus it shares with the processor, it signals the processor asking to use the bus then waits for permission. When permission is given the DMA controller becomes the bus master and locks out the processor for a certain length of time. The problem with this setup is that the bus which the processor is being denied is needed to fetch instructions and data which may not be in the processor's cache memory. This will extend the time it takes to execute code sequences. The costs of DMA must either be eliminated, by choosing not to use DMA, or accounted for as we did with interrupts. There is the option of not implementing cycle stealing DMA and choosing instead to split bus access between the processor and other hardware with a Time Division Multiple Access (TDMA) policy. With TDMA each device which can access the bus is given its fair share of bus cycles. For example, with a system design with one processor and one DMA controller, the bus may be altered between processor and controller every bus cycle. Each device would get 50% of the bus time. This makes the worst case execution of job code long but predictable.

Another source of problems which has to be dealt with is the use of Memory Management Units (MMU) in modern processors. The MMU implements a mapping between virtual and physical addresses that allows an operating system to change the way software sees memory and hardware with respect to their addresses. This mapping is implemented with a set of hierarchical tables which describe the regions of virtual memory and their physical counterparts. These tables are held in physical memory just like application code and data. The MMU has a built in cache called the Translation Lookaside Buffer (TLB) which remembers a subset of the mappings held in memory in order to speed up the translation process. The use of a cache speeds up the translation in most cases but when a translation is needed and the TLB doesn't contain the necessary information then the MMU has to walk through the hierarchy of tables contained in memory which can be costly. Since the MMU is consulted every time instructions or data are fetched from memory the worst case cost has to be considered and modelled. Simply timing your code execution may not give you the worst case execution time since the TLB cache may be non-deterministic — several different executions may give several different timings. One solution is to completely disable the MMU, effectively implementing a one to one mapping, which is easy if you are not using a third party operating system and the hardware allows it. If you cannot do this then you might be able get good worst case execution timings if you can lock down TLB entries with virtual to physical mappings your code will never use. This would give as many TLB misses as possible. Since processor caches behave in a similar way to the TLB, designers may want to turn them off when trying to calculate a job's worst case execution time.

1.5 Which algorithm should I use?

We've seen two scheduling algorithms specifically designed for hard real-time systems: DM which is a static priority algorithm, and EDF which is a dynamic priority algorithm. Scheduling theory says that dynamic priority algorithms can schedule any job set that static priority algorithms can. However, the reverse is not so [But97]. Knowing this, you're probably wondering why we looked at both of them. Surely just using EDF would make things simpler? Yes it would, however, as we saw earlier there are real-time kernels on the market that have first come first served scheduling algorithms. These kernels may be usable in a hard real-time system if the kernel suppliers can provide a worst case execution time for clock interrupt processing, preemption, and any other system calls your application may use. Another advantage to DM is that if the system experiences an overload at a certain priority level then all higher priority jobs will still be scheduled whereas EDF might not preempt the job that is causing the overload.

1.6 The implementation

The implementation of a system using the EVERYMAN kernel is split into three parts. We have the kernel which is written completely in C and is portable across 32-bit platforms. There is the BSP which is platform specific and may have parts written in assembly language. Then we have the application which contains the jobs which the kernel schedules.

To help the system designer build a system based upon the EVERYMAN kernel we'll build a tool which translates a description of the system into a properly configured kernel C code file. This tool is the SDL compiler.

BSP

The BSP will be responsible for configuring and later controlling the hardware used. When the kernel is to be ported to a new platform it will most likely be necessary to rewrite the hardware specific parts.

All five BSP functions start with the prefix `BSP_`:

`BSP_enable_interrupts` When the kernel calls this function the BSP will enable interrupts.

`BSP_deadline_overrun` If this function is called then a job has missed its deadline. This function could be used to try to recover from an overload or even gracefully shutdown the system. The kernel checks the deadlines every clock tick and calls the function if the system time is equal to the absolute deadline of the current job or any pending job.

`BSP_disable_interrupts` When the kernel calls this function the BSP will disable interrupts. This function is called when we update the kernel's internal data structures. When the update is finished the kernel will call the BSP function which enables interrupts.

`BSP_init_exception_vectors` This function will be called by the kernel when the system starts up. This function should maybe be called

`BSP_init_interrupts` but some processor architectures handle hardware interrupts and processor exceptions with the same mechanism. It is often quite useful to be able to handle these exceptions. One example could a divide by zero exception, another could be software breakpoints. It would be up to the BSP to handle such exceptions.

`BSP_init_clock` This function is called by the kernel to instruct the BSP to initialise the real-time clock hardware. It is assumed that this clock generates interrupts and the interrupts will not be generated until the `BSP_enable_interrupts` function is called, that is to say, the `BSP_init_exception_vectors` function will not have interrupts enabled before the first call to `BSP_enable_interrupts`.

The above functions are merely an interface. The designer is obviously free to implement these functions in any way he/she chooses.

Kernel

The kernel also has an API, the functions of which are used to create the system and then respond to the various events generated by either the hardware or the application.

When the hardware starts and has finished its initialisation we expect some piece of BSP code to start executing. This code should call the kernel's initialisation API function so that the hardware is configured in the order that the kernel requires.

Whenever we get a clock interrupt, we need to advance the system time and check for any scheduling activities that need to be performed. The API includes a function for the BSP to call in this case.

In this step we need only one more API function which will create any jobs for the scheduler to process.

So, in total we have three API functions all of which start with `KERN_`.

`KERN_clock_tick` The BSP will call this function every time the real-time clock hardware generates an interrupt. This function will kick the scheduler into action.

`KERN_init` The BSP calls this function after it has loaded the system into memory. This function will create all of the defined jobs and then ask the BSP to initialise exception vectors and the real-time clock hardware. Interrupts will then be enabled and the the job with the highest priority will start running. This function is generated by the SDL compiler.

`KERN_job_create` This function is used to configure the system to schedule a job. It is called by `KERN_init`. The system designer usually does not write code to call this function. It is provided as an API function should you wish to create and use your own system definition mechanism instead of the SDL compiler.

Application

The specifications on the application are not that stringent. Every job has an entrypoint which is a function which is called every time the job moves into the

current state. The designer obviously has to implement a function that will do the work this job has to perform then return within the job's deadline.

There is an optional set of functions in the EVERYMAN kernel called hooks. These functions are called by the kernel at important times during a job activation. The reasoning behind these functions is that it was thought to be important to allow the system designer to log scheduling activity in real-time. The designer may wish to use the functions to track system activity in worst case load conditions. The hook functions are as follows:

`HOOK_job_arrive` Called every time a job arrives.

`HOOK_job_finish` Called every time a job finishes execution.

`HOOK_job_start` Called every time a job actually starts execution, more precisely, this function is called just before a job starts executing its first instruction.

SDL

The EVERYMAN kernel has a System Description Language (SDL) that the designer uses to generate a kernel code file. In this step of the kernel development the SDL compiler may seem like overkill. However, in later steps we see how we can use compile time information to automatically configure the kernel in a way that would be error prone if it were to be done manually.

The SDL compiler is very simple. It takes an input file where the designer specifies three things:

- The scheduling algorithm to be used: EDF or DM.
- Whether hooks are to be used or not.
- The jobs to be scheduled by the system. Each job has a period, a deadline, and an entrypoint function.

With this information the compiler will take a skeleton kernel source code file and add code lines in the beginning of the file which configures the kernel according to the scheduling algorithm used and then at the end of the file the compiler will generate all the calls to the function `KERN_job_create`, one call for each job defined.

The Code

Throughout the code we make use of certain types which are basically synonyms for the basic C types. As you can see in listing 1.1 these are used to describe unsigned 8, 16, or, 32-bit quantities.

Listing 1.1: Misc. types

```
typedef unsigned char U8;
typedef unsigned short U16;
typedef unsigned int U32;
```

Everything we need to know about jobs is described by the types described in listing 1.2. A job is either periodic or sporadic. The fields `period` and `relative_deadline` are the timing parameters T and D. From these two static values we calculate `arrival_time` and `absolute_deadline`. The field `function` is a pointer to a code routine which performs the job's work. The fields `next_job` and `prev_job` are used to keep the `KERN_job_t` on a linked list.

Listing 1.2: Job types

```
typedef enum {
  PERIODIC_JOB,
  SPORADIC_JOB
} KERN_job_type_t;

typedef struct KERN_job KERN_job_t;
struct KERN_job
{
  KERN_job_type_t type;
  U32 relative_deadline;
  U32 absolute_deadline;
  U32 period;
  U32 arrival_time;
  KERN_job_t *next_job;
  KERN_job_t *prev_job;
  void (*function)(void);
};
```

Now we've defined our types we can start to map out the global data that the kernel will need which is given in listing 1.3. The constant `MAX_NUMBER_JOBS` is created by the SDL tool. It is merely the number of jobs which exist in the system. The global variable `job_index` starts at 0 and tells us where the next free space in the array `the_jobs` can be found. See the function `KERN_job_create`.

You can choose not to use the SDL tool and instead define the constant `MAX_NUMBER_JOBS` yourself. Just make sure the array `the_jobs` is big enough for your needs.

Another constant is `MAX_STACK_DEPTH` and again this is defined by the SDL tool. This along with the array stack and the variable stack_index is used to remember what the last job which was current when a preemption occurred. In this step `MAX_STACK_DEPTH` is the same as `MAX_NUMBER_JOBS`. In later steps these constants will be different from each other. See the functions given in listing 1.6 and listing 1.7 for the internal stack API.

The following three variables are used by the scheduler to keep track of the state of all the jobs:

`current_job` The job (periodic or sporadic) that is currently executing.

`pending_jobs_list` A linked list of all the periodic jobs that want to execute.

`inactive_jobs_list` A linked list of all the periodic jobs that await their next period.

The `system_time` starts at zero and is incremented with every clock tick (see listing 1.15). The content of this variable determines the behaviour of the

system. If it is updated irregularly then the computer's idea of time won't match real time. If the variable which is an unsigned 32-bit number wraps around to zero again then you will have problems.

The remainder of listing 1.3 forward defines three functions to avoid compiler warnings.

Listing 1.3: Intro

```
static KERN_job_t the_jobs[MAX_NUMBER_JOBS];
static U32 job_index;
static void *stack[MAX_STACK_DEPTH];
static U32 stack_index;
static KERN_job_t *current_job;
static KERN_job_t *pending_jobs_list;
static KERN_job_t *inactive_jobs_list;
static U32 system_time;

static void job_preempt (void);
static void job_add_to_inactive_jobs_list (KERN_job_t *job);
static void job_arrive (KERN_job_t *job);
```

Listing 1.4 shows the function we call when we want to update the kernel's internal data structures. When we disable interrupts we make preemption impossible.

Listing 1.4: `system_lock`

```
static void system_lock (void)
{
    BSP_disable_interrupts();
}
```

When we've finished updating the kernel's internal data structures we call `system_unlock`.

Listing 1.5: `system_unlock`

```
static void system_unlock (void)
{
    BSP_enable_interrupts();
}
```

The stack which was mentioned earlier is used to save system state we need to "remember" when we preempt jobs. Listings 1.6 and 1.7 are called to save and restore kernel state. In the next step we save more state information than we do in this step.

Listing 1.6: `stack_push`

```
static void stack_push (void *data)
{
    stack[stack_index++] = data;
}
```

Listing 1.7: `stack_pop`

```
static void *stack_pop (void)
```

```
{
    return stack[--stack_index];
}
```

Listing 1.8 shows the function which allocates and initialises a KERN_job_t. Usually this function is used in code generated by the SDL tool. SDL generated calls to KERN_job_create are made before the system starts running. This means interrupts are disabled and system_lock isn't needed to preserve the integrity of the array the_jobs and the variable used to index it: job_index. If you wanted to create jobs manually, possibly dynamically after the system starts, then you would have to surround calls to KERN_job_create with system_lock and system_unlock.

Listing 1.8: KERN_job_create

```
KERN_job_t *KERN_job_create (U32 period, U32
    relative_deadline, void (*function)(void),
    KERN_job_type_t type)
{
  KERN_job_t *job;

  if(job_index > MAX_NUMBER_JOBS - 1)
    {
      return NULL;
    }

  job = &the_jobs[job_index++];
  job->type = type;
  job->period = period;
  job->relative_deadline = relative_deadline;
  job->function = function;

  if(job->type == PERIODIC_JOB)
    {
      job->absolute_deadline = system_time + job->
          relative_deadline;
      job_arrive(job);
    }
  return job;
}
```

Whenever a job's period starts then it is moved into the pending state. The function job_add_to_pending_jobs_list shown in listing 1.9 does just that. The pending jobs list is sorted in order of priority, the head of the list having highest priority.

Listing 1.9: job_add_to_pending_jobs_list

```
static void job_add_to_pending_jobs_list (KERN_job_t *job)
{
    KERN_job_t *runner;
    KERN_job_t *last_runner;

    for(last_runner=NULL, runner = pending_jobs_list; runner
        ; last_runner=runner, runner = runner->next_job)
    {
```

```
#if defined OPT_EDF_SCHEDULING
        if(runner->absolute_deadline > job->
            absolute_deadline)
#elif defined OPT_DM_SCHEDULING
        if(runner->relative_deadline > job->
            relative_deadline)
#else
#error "no scheduler defined"
#endif
        {
            break;
        }
    }
    if(runner)
    {
        job->prev_job = runner->prev_job;
        if(runner->prev_job)
        {
            runner->prev_job->next_job = job;
        }
        else
        {
            pending_jobs_list = job;
        }

        job->next_job = runner;
        runner->prev_job = job;

    }
    else
    {
        if(last_runner)
        {
            last_runner->next_job = job;
            job->prev_job = last_runner;
            job->next_job = NULL;
        }
        else
        {
            job->prev_job = job->next_job = NULL;
            pending_jobs_list = job;
        }
    }
}
```

The internal function `job_arrive` in listing 1.10 is used to get a job onto the pending list. We need to do this when either a periodic job is created (listing 1.8), or a periodic job's period has started (listing 1.15), or a sporadic job has been released (listing 1.16). Since sporadic jobs can't have their absolute deadlines calculated statically, it is updated here. See listing 1.12 for the periodic job's absolute deadline calculation.

Listing 1.10: `job_arrive`

```
static void job_arrive (KERN_job_t *job)
```

```
{
  if(job->type == SPORADIC_JOB)
    {
      job->absolute_deadline = system_time + job->
          relative_deadline;
    }
  job_add_to_pending_jobs_list(job);

#ifdef OPT_USE_HOOKS
    HOOK_job_arrive(job);
#endif
}
```

The function job_add_to_inactive_jobs_list is shown in listing 1.11. This function is called when a job has finished execution and we want it to wait until the start of its next period. This function should not be used with sporadic jobs. See the function job_finish (listing 1.12) for the only caller of this function.

Listing 1.11: job_add_to_inactive_jobs_list

```
static void job_add_to_inactive_jobs_list (KERN_job_t *job)
{
    KERN_job_t *runner;
    KERN_job_t *last_runner;

    for(last_runner=NULL, runner = inactive_jobs_list;
        runner; last_runner = runner, runner = runner->
        next_job)
    {
        if(runner->arrival_time > job->arrival_time)
        {
            break;
        }
    }
    if(runner)
    {
        job->prev_job = runner->prev_job;
        if(runner->prev_job)
        {
            runner->prev_job->next_job = job;
        }
        else
        {
            inactive_jobs_list = job;
        }

        job->next_job = runner;
        runner->prev_job = job;

    }
    else
    {
        if(last_runner)
        {
```

```
            last_runner->next_job = job;
            job->prev_job = last_runner;
            job->next_job = NULL;
        }
        else
        {
            job->prev_job = job->next_job = NULL;
            inactive_jobs_list = job;
        }
    }
}
```

Listing 1.12 shows `job_finish` which is called when a job finishes execution. here we set up the job's timing parameters for the next period.

Listing 1.12: `job_finish`

```
static void job_finish (KERN_job_t *job)
{
  if(job->type == PERIODIC_JOB)
    {
      job->arrival_time = (job->arrival_time + job->period);
      job->absolute_deadline = job->arrival_time + job->
          relative_deadline;
      job_add_to_inactive_jobs_list(job);
    }
#ifdef OPT_USE_HOOKS
    HOOK_job_finish(job);
#endif

    current_job = stack_pop();
}
```

Listing 1.13 shows the library function `job_list_unlink` which removes the job at the head of the `job_list`. This is currently only used on the pending jobs list.

Listing 1.13: `job_list_unlink`

```
static KERN_job_t *job_list_unlink (KERN_job_t **job_list)
{
    KERN_job_t *retval;

    if(*job_list)
    {
        retval = *job_list;
        if(retval->next_job)
        {
            retval->next_job->prev_job = NULL;
        }
        *job_list = retval->next_job;
    }
    else
    {
        retval = NULL;
    }
```

```
    return retval;
}
```

Listing 1.14 shows the internal function job_preempt. This function is called to check for any possible preemption. The purpose is basically to find the job with the highest priority which is either currently executing or pending. If this job isn't the currently executing job then we make sure that it becomes the currently executing job.

Listing 1.14: job_preempt

```
static void job_preempt (void)
{
  system_lock();
  for(;;)
    {
      if(pending_jobs_list)
        {
          if(!current_job ||
#if defined OPT_EDF_SCHEDULING
             (current_job->absolute_deadline >
                pending_jobs_list->absolute_deadline))
#elif defined OPT_DM_SCHEDULING
             (current_job->relative_deadline >
                pending_jobs_list->relative_deadline))
#else
#error "no␣scheduler␣defined"
#endif
            {
              /* We need to preempt the currently executing
                  job */

              /* Remember the current job so we can return
                  to executing it later */
              stack_push(current_job);

              /* Get the new current job */
              current_job = job_list_unlink(&
                  pending_jobs_list);

#ifdef OPT_USE_HOOKS
              HOOK_job_start(current_job);
#endif
              system_unlock();

              /* Start executing the new current job */
              current_job->function();

              system_lock();
              job_finish(current_job);
            }
          else
            {
              /* No preemption - the current job has the
                  highest priority.*/
```

```
              break;
            }
        }
      else
        {
          /* No preemption - there are no pending jobs */
          break;
        }
    } /* for(;;) */
  system_unlock();
}
```

The core of system scheduling is implemented in the function KERN_clock_tick shown in listing 1.15. This function is called at every system clock tick. The system_lock function prevents this function from executing and as a consequence protects all of the kernel's internal data structures from corruption.

Listing 1.15: KERN_clock_tick

```
void KERN_clock_tick (void)
{
  KERN_job_t *j, *temp;

  system_lock();
  system_time++;

  /* Has the current job missed its deadline */
  if(current_job && system_time == current_job->
      absolute_deadline)
    {
      BSP_deadline_overrun(current_job);
    }

  /* Have any pending jobs missed their deadlines */
  for(j = pending_jobs_list; j; j=j->next_job)
    {
      if(system_time == j->absolute_deadline)
        {
          BSP_deadline_overrun(j);
        }
    }

  /* See if any jobs have started their periods */
  for(j = inactive_jobs_list; j;)
  {
      if(system_time == j->arrival_time)
      {
          temp = j;
          j=j->next_job;
          if(temp == inactive_jobs_list)
          {
              inactive_jobs_list = j;
              if(j)
              {
                  j->prev_job = NULL;
```

```
                }
            }
            else        /* shouldn't need this "else" */
            {
                temp->prev_job->next_job = j;
                if(j)
                {
                    j->prev_job = temp->prev_job;
                }
            }
            /* Move the job into the pending jobs list */
            job_arrive(temp);
        }
        else
        {
            j=j->next_job;
        }
    }
    system_unlock();

    /* We may have a possible preemption */
    job_preempt();
}
```

KERN_job_arrive (listing 1.16) is called in order to get a sporadic job into the pending jobs list. Don't call this function on periodic jobs or when the interrupts are disabled. If you really need to move a job into the pending state and interrupts are disabled, then use the internal function job_arrive (listing 1.10) instead and when you enable interrupts again call job_preempt.

Listing 1.16: KERN_job_arrive

```
void KERN_job_arrive (KERN_job_t *job)
{
    system_lock();
    job_arrive(job);
    system_unlock();
    job_preempt();
}
```

This completes the design of the kernel for Step 1.

STEP 2

Here we build upon the previous step and introduce a resource access protocol which will eliminate unbounded blocking and priority inversion.

The previous step gave us the ability to have several jobs on the go at the same time. In order to perform useful work, a job will need access to things like hardware registers or data structures in memory which are often needed by other jobs. We model these registers and data structures as resources and control the access to the resources with a protocol

We can denote a resource with the symbol R. Each resource will have a maximum of N_R units available and we can denote the current amount of units available for a resource with ν_R. It should come as no surprise that $0 \leq \nu_R \leq N_R$.

2.1 Blocking

When a job wants a certain number of units of a certain resource then it performs a request. We write $\mu_R(J)$ to denote the number of units of resource R that the job J will request.

If $\mu_R(J) > \nu_R$ then the resource R does not have enough units to satisfy the request so J would have to wait for the required number of units of the resource to become available, i.e. wait until $\mu_R(J) \leq \nu_R$. When a job has to wait it is said to block.

When there are enough units available then these units are allocated to the job. After the job has finished performing the work it needs to do with the resource, it releases a certain number of units of the allocated resource. The most common case would be that the job releases exactly the same amount of units as it requested.

A mutex is an example of a single unit resource ($N_R = 1$). A reader/writer lock is an example of a multiple-unit resource. In this example, the value of N_R would have to be greater than or equal to the number of resource users. A reader would request one unit and a writer would request all of the units. This means that if any job was reading then all writers would be blocked since $\nu_R < N_R$, and if there was one writer then all other readers and writers would block since $\nu_R = 0$.

Let's take a single unit resource R and two jobs, J_1 and J_2, which need to use this resource. If both jobs need to use this resource at the same time then one of them will have to be blocked. Figure 2.1 shows the case where J_2 is allocated the resource and J_1 blocks waiting for J_2 to finish. This could happen if J_2 was the current job and just after it was allocated the resource, it was preempted by J_1. If we schedule the jobs according to RM or EDF, and we do not allow resources to be held across periods, then we know that J_1 will have an upper bound on the amount of time it will be blocked. In the worst case this bound will be the time from J_1's request up to the absolute deadline of J_2. Not surprisingly we say that this is a bounded blocking.

To give an example of unbounded blocking, we expand our example and introduce a new resource R_2 which both J_1 and J_2 want to use. Imagine the jobs were scheduled as follows:

Figure 2.1: Resource blocking.

Step 1 J_1 becomes the current job and its code starts to execute. It requests R_1 and has the resource allocated to it.

Step 2 J_2 now becomes pending, preempts J_1, requests and is allocated R_2.

Step 3 J_2 requests R_1 which is not available and it blocks. The job is moved from the current state to allow another job to make progress.

Step 4 J_1 becomes the current job again and it requests R_2 which is not available and it blocks.

In this example J_1 must block because it needs the resource J_2 has. J_2 also must block because it needs the resource J_1 has. Both of these jobs are said to be deadlocked.

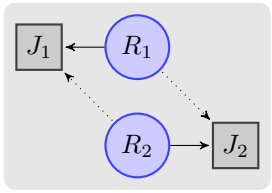


Figure 2.2: Resource deadlock.

One proposed solution to deadlock could be to let deadlocked jobs release their resources and start all over again. In this case J_1 would release R_1, J_2 would release R_2, and the jobs would somehow be moved back into the execution state before they tried to request resources. The idea behind this proposal is that the jobs probably wouldn't be deadlocked the next time they requested their resources thus solving the problem. The thing is that system designers have seen problems with such a solution. It is not unheard of for systems to become caught up in a deadlock-release loop — once the resources were released the scheduler executed the jobs in such a way that the jobs became deadlocked again. This is known as a livelock. Both livelock and deadlock are extreme examples of unbounded blocking.

Here is another example of a blocking problem that is quite common. Suppose we have three jobs: J_1, J_2, and J_3. J_1 has a high priority, J_3 has a low priority, and J_2 has a priority somewhere between J_1 and J_3. J_1 and J_3 both need access to the same resource. Imagine J_3 is the current job and it requests and is allocated the resource it needs. Before it is finished with the resource J_1 preempts it and requests the same resource which leads to a blocking. The question is how long will J_1 have to wait until it is unblocked? The answer depends on the execution times of both J_3 and J_2 since J_2 can preempt J_3 and

prevent it from finishing its work with the resource J_1 is blocked waiting for. This situation is known as priority inversion since the scheduler is letting J_2 make progress at the expense of J_1 which has a higher priority. If the processor was allocated to J_3 instead, then it would mean J_1 would be blocked for the minimum amount of time which is obviously desirable. A solution to this would be to prevent the preemption of J_3 by any job with a priority less than or equal to J_1 whilst J_1 was blocked. This would effectively let J_3 inherit J_1's priority whilst J_3 was blocked.

2.2 Stack Resource Policy

A mechanism for resource allocation which eliminates deadlock, livelock, and priority inversion was invented in 1990. It is called the stack Resource Policy (SRP) [Bak91]. Essentially, this policy will not allow jobs to enter the current state if all of the resources they need are not available. This means once a job starts it will have no reason to block which eliminates deadlock and livelock. The policy also makes sure that a job which holds a resource will effectively inherit the priority of any higher priority job which it is preventing from entering the current state. This eliminates priority inversion.

We limit resource request and release to be performed in Last In First Out (LIFO) order. This means that if a job requests R_1 then R_2, it must release the resources in the order R_2 then R_1. We will also say that jobs cannot hold resources across job periods — jobs must request and release all resources within a period.

Job preemption level

With SRP, every job is given a preemption level. For a job J the preemption level is denoted $\pi(J)$. The fundamental rule is that a job may only preempt another job if it has both a higher priority and a higher preemption level. We set a job's preemption level to be inversely proportional to the relative deadline. With DM the preemption level is exactly the same as the priority so this fundamental rule obviously holds. With EDF the rule still holds but it is not as obvious to see why. Looking at figure 2.3 we see on the top timing diagramme that J_2 may have a higher preemption level but during J_2's period it does not have a higher priority, so there is no preemption. On the bottom timing diagramme J_2 will preempt J_1 because it has both a higher priority and a higher preemption level. The SRP rule regarding preemption levels holds with EDF because a preempting job always starts after the job it preempts and it must have a shorter relative deadline to have a higher priority (shorter absolute deadline). The job preemption level is a mechanism used by the SRP to turn the EDF dynamic priorities into a static value.

Resource ceiling

Each resource has a preemption ceiling denoted by $\overline{R}_{\nu_R}$. The subscript ν_R is the number of available units of resource R. We need this subscript because, and you will later see why, the ceiling can change when ν_R changes. The definition of a resource's preemption ceiling is:

$$\overline{R}_{\nu_R} = \max(\{0\} \cup \{\pi(J) | \nu_R < \mu_R(J)\})$$

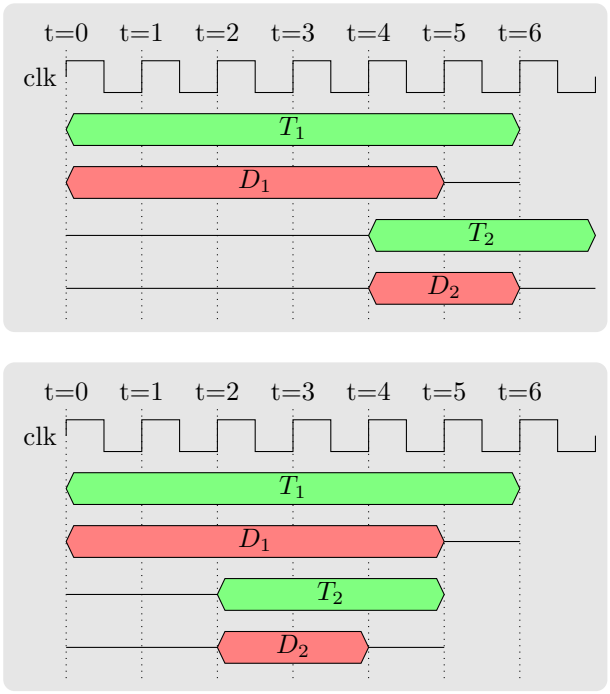


Figure 2.3: EDF preemption levels.

Earlier we defined $\mu_R(J)$ as the amount of resource R that job J will request, and if $\nu_R < \mu_R(J)$ then the job would need to block. So, the resource ceiling $\overline{R}_{\nu_R}$ is equal to the highest preemption level of all jobs that will block when resource R has only ν_R units available. If no job would block then $\overline{R}_{\nu_R}$ is zero.

We next define a system ceiling $\overline{\pi}$ which is defined to be the maximum of all the current ceilings of all the resources. We set $\overline{\pi}$ to be initially zero. We make sure that no job J is allowed to move into the current state unless $\pi(J) > \overline{\pi}$.

Every time that a resource R is requested the value of ν_R is updated to reflect that fact that some units are no longer available. $\overline{\pi}$ is pushed onto a stack and it is then compared to $\overline{R}_{\nu_R}$. If it so happens that $\overline{R}_{\nu_R} > \overline{\pi}$ then $\overline{\pi}$ is set to the value of $\overline{R}_{\nu_R}$. Otherwise, $\overline{\pi}$ remains unchanged.

When a resource is released then ν_R is updated to reflect that more units are now available and $\overline{\pi}$ is set to a value that is popped from the stack. Effectively, this restores the value of $\overline{\pi}$ to that which it had before the resource was allocated. This stack correctly tracks the value of $\overline{\pi}$ because we limited resource requests and releases to be performed in LIFO order.

As an example of how job preemption levels give resource ceiling tables we will look at an example of a reader/writer lock. We will have three jobs, J_1 and J_2 are readers, and J_3 is a writer. The table 2.1 shows the jobs and their preemption levels and how many units of the resource they request. The resource ceiling table is shown in table 2.2. Since we have three jobs we set $N_R = 3$.

When all units are available any job can be moved into the current state. Should the writer job, J_3 make its request then $\overline{\pi}$ would become $\overline{R}_0 = 3$ which prevents the other jobs from moving into the current state. If, on the other hand, one of the reader jobs made a request then $\overline{\pi}$ would become $\overline{R}_2 = 1$. In this case, the writer job, J_3 cannot move into the current state as $\overline{\pi} \geq \pi(J_3)$. On the other hand it is still possible for the other reader to move into the current state as both $\pi(J_1)$ and $\pi(J_2)$ are greater than $\overline{\pi}$. This is shown visually in figure 2.4 where on the left we have the writer preventing the readers, and, on the right we see the readers preventing the writer.

If the writer did not have the lowest preemption level like in this example then the reader/writer lock wouldn't work as we want it to. The first reader which made a request would raise $\overline{\pi}$ to a level higher than any reader which had a preemption level lower than a writer.

Table 2.1: SRP example job set.

J	$\pi(J)$	$\mu_R(J)$
J_1	3	1
J_2	2	1
J_3	1	3

Table 2.2: Resource ceiling table.

ν_R	$\overline{R}_{\nu_R}$
0	3
1	1
2	1
3	0

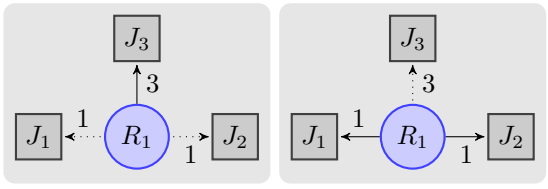


Figure 2.4: SRP example.

2.3 Feasibility Tests

The maximum blocking time experienced by a job is the time that a lower priority job may hold a resource. A job can only be blocked by only one lower priority job. To calculate the blocking time, B_i, for job J_i we will obviously have to time every code sequence between a resource request and release for all jobs which have a preemption level lower than $\pi(J_i)$

As in Step 1, we assume we have a set of n jobs, $\{J_1, ..., J_n\}$, where J_1 has the highest priority and J_n the lowest.

The feasibility test for Deadline Monotonic is updated to include the blocking time in the response time analysis.

$$R_i = C_i + B_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k + \sum_{j=1}^{m} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

With Earliest Deadline First the change is not so simple. The test including blocking is:

$$\forall L \in \mathbb{D}, 1 \leq i \leq n, L - \mathrm{f}(L) \geq \sum_{k=1}^{i} \left(\left\lfloor \frac{L - D_k}{T_k} \right\rfloor + 1 \right) C_k + \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) B_i$$

Instead of calculating the processor demand for all jobs at every absolute deadline, we need to take each job in turn and calculate the processor demand for that job and all jobs with higher preemption levels, plus, the blocking time from jobs with lower preemption levels.

- For each job

$$\forall L \in \mathbb{D}, \; 1 \leq i \leq n, \; L - \mathrm{f}(L) \geq \sum_{k=1}^{i} \left(\left\lfloor \frac{L - D_k}{T_k} \right\rfloor + 1 \right) C_k + \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) B_i$$

- Processor demand for J_i and all jobs with higher preemption level
- Total blocking time from lower preemption level jobs

The reason we have to analyse each job in turn is that blocking time isn't lost time. Let's say we have a job set and there is a job J_i can be blocked by J_j for B_i units of time. If we try to perform a processor demand analysis for all jobs like we did in the last step and somehow work in the blocking times for each job, then we would end up including the blocking times twice. In our example we would be using both B_i and the whole of C_j which also includes B_i. This is obviously too pessimistic.

2.4 Implementation

As we know, each job will now have a preemption level associated with it ($\pi(J)$). The kernel's job data type, which is shown in listing 2.1, is changed to include a field called `preemption_level`.

Listing 2.1: `KERN_job_t`

```
typedef struct KERN_job KERN_job_t;
struct KERN_job
{
  KERN_job_type_t type;
  U32 preemption_level;
```

```
  U32 relative_deadline;
  U32 absolute_deadline;
  U32 period;
  U32 arrival_time;
  KERN_job_t *next_job;
  KERN_job_t *prev_job;
  void (*function)(void);
};
```

Resources are described by the data type `KERN_resource_t` shown in listing 2.2. The field `available` is ν_R, and, the field `ceiling` is $\overline{R}$.

Listing 2.2: `KERN_resource_t`

```
typedef struct
{
  U32 available;
  U32 *ceiling;
} KERN_resource_t;
```

Listing 2.3 shows the memory which is statically allocated to hold resource definitions. The maximum number of resources that jobs can use is determined by `MAX_NUMBER_RESOURCES`.

This constant is determined by the SDL tool. The array `the_resources` holds the details of all the defined resources. The variable `resource_index` is used to find the next entry in `the_resources` which has not been allocated. See listing 2.7 for the creation of resources.

Listing 2.3: Resource globals

```
static KERN_resource_t the_resources[MAX_NUMBER_RESOURCES];
static U32 resource_index;
```

The SRP algorithm requires a system ceiling, $\overline{\pi}$, which is implemented as a global variable shown in listing 2.4.

Listing 2.4: `system_ceiling`

```
static U32 system_ceiling;
```

Moving on to the code which implements the behaviour of the SRP algorithm, we need to provide functions for requesting and releasing the resources that applications can use; we also need to be able to initialise a variable of the type `KERN_resource_t`, and we need to alter the code which handles possible preemptions so that the condition $\pi(J) \geq \overline{\pi}$ is true for any job J which wants to preempt the current job.

The first function we'll look at is `KERN_resource_request` shown in listing 2.5. The operation is quite simple, we save information on the stack that will be restored when the resource is released with `KERN_resource_release`. We finally update $\nu(R)$, $\overline{R}_{\nu_R}$, then possibly $\overline{\pi}$. This function cannot lead to a preemption since $\overline{\pi}$ is never lowered.

In the previous step we discussed the dimension of the state stack. We see that in this step we have increased the amount of information we push and pop. You can see that we push information onto the stack in `KERN_resource_request`, and pop information off the stack in `KERN_resource_release`. The stack works just fine for us as all resource requests and releases are in LIFO order. Job

code is constrained to release all requested resources within the time frame of one job activation so the use of the stack with resources will not be in conflict with the stack pushes and pops we described in the previous step. The only question that remains is how to properly dimension the stack for the worst case scenario. The SDL tool sets the constant `MAX_STACK_DEPTH` to be three times the number of resources plus the number of jobs. This dimension covers maximum preemption and one request of each resource with no multiple requests of each resource outstanding, i.e, the resources are being used as mutexes. If this is not the case then you may want three times the number of resources times the number of jobs which would cover the resources being used as counting semaphores with a job only requesting a given resource once during its period. You will have to dimension this stack to suit your own needs.

Listing 2.5: `KERN_resource_request`

```
void KERN_resource_request (KERN_resource_t *res, U32 amount
    )
{
    U32 resource_ceiling;

    system_lock();

    stack_push((void *)system_ceiling);
    stack_push((void *)res->available);
    stack_push(res);

    res->available = res->available - amount;
    resource_ceiling = res->ceiling[res->available];
    if(resource_ceiling > system_ceiling)
    {
        system_ceiling = resource_ceiling;
    }

    system_unlock();
}
```

The function `KERN_resource_release` shown in listing 2.6 restores the SRP state to that which it was before the most previous `KERN_resource_request`. Here $\overline{\pi}$ can be lowered which means that a preemption is possible.

Listing 2.6: `KERN_resource_release`

```
void KERN_resource_release (void)
{
    KERN_resource_t *res;
    U32 old_system_ceiling;

    system_lock();

    res = stack_pop();
    res->available = (U32)stack_pop();
    old_system_ceiling = system_ceiling;
    system_ceiling = (U32)stack_pop();

    if(system_ceiling < old_system_ceiling)
```

```
    {
        system_unlock();
        job_preempt();
    }
    else
    {
        system_unlock();
    }
}
```

Now we look at the function `KERN_resource_create` (listing 2.7) which is used to initialise variables of the type `KERN_resource_t`. The `ceiling` field is a pointer to an array of 32-bit unsigned integers. In the function `KERN_job_create` (listing 2.8) we see that $\forall i, \pi(J_i) = 0xffffffff - D_i$. Here we assume the processor running the kernel is 32-bit. Since these preemption levels, $\pi(J)$, make up the ceiling table entries, we have to be careful that we calculate them in the same way.

The SDL tool automatically creates the ceiling tables which are passed to `KERN_resource_create`. Just like the case with calls to `KERN_job_create`, the resources are created before scheduling starts. This means that the system is essentially locked and the kernel can be sure that any code that executes will not be interrupted. If you wanted to call `KERN_resource_create` from your own code then it would be wise to surround the function call with calls to `KERN_system_lock` and `KERN_system_unlock`. You would also need to ensure that the constant `MAX_NUMBER_RESOURCES` was dimensioned properly.

Listing 2.7: `KERN_resource_create`

```
KERN_resource_t *KERN_resource_create(U32 available, U32 *
    ceiling)
{
  KERN_resource_t *res;

  if(resource_index > MAX_NUMBER_RESOURCES - 1)
    {
      return NULL;
    }

  res = &the_resources[resource_index++];
  res->available = available;
  res->ceiling = ceiling;

  return res;
}
```

The function responsible for job creation (listing 2.8) is updated to give each job its preemption level — $\pi(J)$. We assume a 32-bit processor when we define the `preemption_level` to be `0xffffffff - relative_deadline`.

Listing 2.8: `KERN_job_create`

```
KERN_job_t *KERN_job_create (U32 period, U32
    relative_deadline, void (*function)(void),
    KERN_job_type_t type)
{
```

```
  KERN_job_t *job;

  if(job_index > MAX_NUMBER_JOBS - 1)
    {
      return NULL;
    }

  job = &the_jobs[job_index++];
  job->type = type;
  job->period = period;
  job->relative_deadline = relative_deadline;
  job->function = function;
  job->preemption_level = 0xffffffff - relative_deadline;

  if(job->type == PERIODIC_JOB)
    {
      job->absolute_deadline = system_time + job->
          relative_deadline;
      job_arrive(job);
    }
  return job;
}
```

Finally, we only have to make changes to the preemption logic so that the SRP condition that $\pi(J) \geq \overline{\pi}$ always holds. Listing 2.9 shows the new version of `KERN_job_preempt`. The only addition is just this SRP preemption condition: `(system_ceiling < pending_jobs_list->preemption_level)`.

Listing 2.9: `KERN_job_preempt`

```
static void job_preempt (void)
{
  system_lock();
  for(;;)
    {
      if(pending_jobs_list)
        {
          if(!current_job ||
#if defined OPT_EDF_SCHEDULING
             ((current_job->absolute_deadline >
                pending_jobs_list->absolute_deadline) &&
#elif defined OPT_DM_SCHEDULING
             ((current_job->relative_deadline >
                pending_jobs_list->relative_deadline) &&
#else
#error "no␣scheduler␣defined"
#endif
              (system_ceiling < pending_jobs_list->
                 preemption_level)))
            {
              /* We need to preempt the currently executing
                 job */

              /* Remember the current job so we can return
                 to executing it later */
```

```
              stack_push(current_job);

              /* Get the new current job */
              current_job = job_list_unlink(&
                  pending_jobs_list);

#ifdef OPT_USE_HOOKS
              HOOK_job_start(current_job);
#endif
              system_unlock();

              /* Start executing the new current job */
              current_job->function();

              system_lock();
              job_finish(current_job);
            }
          else
            {
              /* No preemption - the current job has the
                  highest priority.*/
              break;
            }
        }
      else
        {
          /* No preemption - there are no pending jobs */
          break;
        }
    } /* for(;;) */
  system_unlock();
}
```

STEP 3

In this step we'll introduce a communications primitive which can be used by jobs to share information with each other.

Most systems will consist of a collection of cooperating jobs. To facilitate co-operation, the kernel will have to provide mechanisms for safe synchronisation and communication. The SRP algorithm will take care of the synchronisation so we'll concentrate on communication in this step.

If we looked at every kernel ever written and all of the communications primitives that they provided we would see that these primitives can be roughly divided into two groups: shared memory and distributed memory.

A shared memory primitive would allow our jobs to safely access common areas of memory where they could all read and write. The SRP algorithm which we saw in the last step is a type of shared memory communication primitive. So we won't discuss shared memory primitives any more here.

A distributed memory primitive on the other hand would allow our jobs to send data to, or receive data from, other jobs. You can think of this as jobs distributing their data to other jobs. The most common model for distributed memory primitives is the message passing paradigm which comprises two operations: send and receive. When we look at various implementations of these operations we see that they can be synchronous or asynchronous. With communication primitives, synchronous means "may possibly block", whereas asynchronous means "will never block".

If we receive messages synchronously then we'll have to block if there is no message to be read. Imagine no job has sent us a message so we're forced to wait until the first message arrives. Should there be a message ready to be received then the primitive will immediately return with a message.

When sending messages synchronously we can block until the the job we are sending to receives the message. This synchronous send is sometimes called a rendezvous and it allows jobs to both communicate and synchronise. The synchronisation aspect is useful as each job knows the general status of the other when they start to execute instructions after the send or receive calls.

With regards to real-time systems and synchronous communications, the main worry is trying to bound the amount of time jobs are blocked. If we cannot calculate an upper bound then there is no way to analyse the feasibility of a job set and this means we'll be forced into using asynchronous operations.

The way that asynchronous primitives are implemented differ from system to system but we can easily identify two implementation flavours.

One way to implement send and receive is as function calls which return to the caller with some indication of success. The function which sends messages would always return a success code (unless there was not enough room to hold data). The function used to receive data could possibly return an error code indicating that there was no message to be received. This type of implementation is very much in the spirit of traditional application program design.

Another way of implementing asynchronous send and receive operations is to have the job register callback functions which would be executed when the send or receive had been successfully completed. This type of implementation would lead to a more event-driven type of system design. It is almost as if communication was modelled in terms of software interrupts.

3.1 Cyclical Asynchronous Buffers

The year 1989 saw the introduction of Cyclical Asynchronous Buffers (CAB) [Cla89]. A CAB is an asynchronous distributed memory communications mechanism. Furthermore, communication is between one sender and several receivers, i.e. it is a broadcast mechanism. Figure 3.1 shows a CAB C which has one sending job J_1, and two receiving jobs J_2 and J_3.

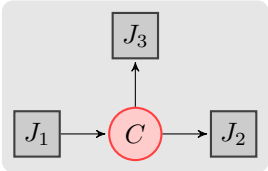


Figure 3.1: A Simple CAB example.

A CAB can be thought of as a communications channel which is used to send and receive buffers. We'll use the term buffer and not signal because the CAB's communication semantics are not the same as those of traditional signal primitives.

With standard signal reception, the signal is consumed — a signal can be received only once. A buffer sent with a CAB can be received any number of times. Actually, it may be the case that a buffer is never be received. Whenever a job receives a buffer from a CAB it is given the last buffer sent to the CAB. A job which receives buffers at a higher frequency than they are sent will receive the same buffer several times. If buffers are sent at a higher frequency than they are received then some buffers will be skipped over and never received. If several jobs try to receive a buffer at the same time then they may actually get the same buffer.

These semantics won't work for some applications but it does allow the sending job and all the receiving jobs to have independent periods of activation.

To give an example, imagine we have three environmental variables controlled by a system, that is to say three "things" in the environment which can be sampled with sensors or manipulated with servos. Let's create a job for each variable and call them J_1, J_2 and, J_3. The sole purpose of each job is to make decisions with regards to what kind of manipulation is needed to keep the variable controlled. In some cases it might be useful to have a higher level job which tried to control the system as a whole. Think of J_1, J_2 and, J_3 making local decisions and a new job J_4 making global decisions. Figure 3.2 shows this graphically.

If J_4 is going to make good decisions then it needs to be updated when the state of the world changes. The obvious design is to let the local decision makers inform J_4 whenever the state of their little part of the world is altered. In the worst case J_4 will need to be communicated with at each activation period of J_1, J_2 and, J_3. Looking at table 3.1 we see that if we used traditional signals then J_4 would need quite a large number of signal buffers in its input queue, and, it would have to spend a lot of time receiving messages every period in order to avoid an overflow of signal buffers. If CABs were used on the other

hand then only three CABs with three buffers each would be needed and J_4 would need to receive a maximum of three buffers during its period.

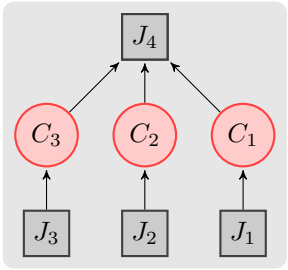


Figure 3.2: Using a CAB for multilevel control.

Table 3.1: Multilevel control CAB example.

Job	Function	Frequency
J_1	Local decision maker	200Hz
J_2	Local decision maker	150Hz
J_3	Local decision maker	10Hz
J_4	Global decision maker	1Hz

The CAB manages the memory used for the buffers. This means that when the CAB is created then it has to be given an amount of memory which will cover the application's needs.

CABs are implemented in a traditional function call model, jobs call CAB functions which return when the operation is completed.

When we want to send a message with a CAB we first need to reserve a buffer. This operation should return a buffer from the CAB's free list. After the buffer has been written to, we then use the put operation which makes that buffer the buffer the most recently written to. The previously most recent buffer is returned to the free list if there is no job reading from it.

1. "Reserve" a free buffer.
2. Copy data to be sent into the buffer.
3. "Put" the buffer

When we receive from a CAB we use the get operation to give us the most recently written to buffer. When we have finished reading we use the unget operation which checks to see if the buffer is still the most recently written to. If it is not and there is nobody else is reading the buffer then we can move the buffer to the free list.

1. "Get" the most recently written to buffer.
2. Copy/process data contained within the buffer.
3. "Unget" the buffer.

Now we know how to read and write buffers we can consider how to dimension the CAB so that there is always a buffer on the free list when we reserve so that operation will never block. A get will never block if it follows a put so we need one write before any reads. Either that or we return a special value to signify no buffer has been written to. The number of buffers we need is equal to one plus the number of concurrent read and writes that can be ongoing at any one time.

3.2 Implementation

The implementation of the CAB primitives is fairly independent of the kernel. The only functions that are needed are `system_lock` and `system_unlock`.

When trying to understand the code, the place to look at first is the data structure used to represent a CAB. Listing 3.1 shows two data structures: `KERN_cab_buffer` and `KERN_cab_t`. The structure used to represent messages that jobs will send to each other is `KERN_cab_buffer`. Here is a description of what each part of that structure is used for:

`next` This is used to link free (unused) buffers on a special linked list.

`use_counter` The number of jobs receiving this buffer is held here.

`data` This is the buffer's payload.

All of the bookkeeping associated with a CAB is stored in the structure `KERN_cab_t`:

`number_of_buffers` How many buffers are connected to this CAB.

`most_recent` The last buffer that was sent (put).

`free_list` This is the linked list of buffers which are free to be allocated with `KERN_cab_reserve`.

Listing 3.1: `KERN_cab_t` and `KERN_cab_buffer_t`

```
typedef struct KERN_cab_buffer KERN_cab_buffer_t;
struct KERN_cab_buffer
{
  KERN_cab_buffer_t *next;
  U32 use_counter;
  U8 data[1];
};

typedef struct
{
  U32 number_of_buffers;
  KERN_cab_buffer_t *most_recent;
  KERN_cab_buffer_t *free_list;
} KERN_cab_t;
```

To receive a message from the CAB we use the functions `KERN_cab_get` and `KERN_cab_unget`.

We start with KERN_cab_get shown in listing 3.2. Here we see the usual surrounding calls to system_lock and system_unlock to make the function reentrant. The meat of the function is simply concerned with returning a pointer to the most recently sent buffer. If no buffer has been sent, then we can only return a NULL pointer and hope the calling job can handle this case. Since we're implementing an asynchronous communications primitive, we cannot block. If there is a most recently sent buffer then we increment the buffers use_counter field which tells the system how many jobs are receiving this buffer. You'll see this counter being used in the function KERN_cab_unget (listing 3.3).

Listing 3.2: KERN_cab_get

```
KERN_cab_buffer_t * KERN_cab_get(KERN_cab_t *cab)
{
  KERN_cab_buffer_t *buffer;

  system_lock();
  buffer = cab->most_recent;
  if(!buffer)
    {
      goto out;
    }
  buffer->use_counter++;
 out:
  system_unlock();
  return buffer;
}
```

Moving on to KERN_cab_unget shown in listing 3.3 we see the latter half of message reception. Again we see this is a reentrant function as we are using system_lock and system_unlock. Between these calls we check to see if the buffer which has been received has no other receivers and it is not the most recently sent buffer. If this is the case then it is safe to recycle this buffer by moving it onto the CAB's free buffer list.

Listing 3.3: KERN_cab_unget

```
void KERN_cab_unget(KERN_cab_t *cab, KERN_cab_buffer_t *
    buffer)
{
    system_lock();
    if(!--buffer->use_counter && buffer != cab->most_recent)
    {
        buffer->next = cab->free_list->next;
        cab->free_list = buffer;
    }
    system_unlock();
}
```

When we want to send a message we call KERN_cab_reserve (listing 3.4), fill in the payload details and when we are ready we call KERN_cab_put (listing 3.5)

Reserving a buffer is easy. We take the first element of the free list. If this list is empty then we have incorrectly dimensioned the buffer (see listing 3.6). The allocated buffer, or NULL in the case of error, is returned to the caller.

Listing 3.4: KERN_cab_reserve

```
KERN_cab_buffer_t * KERN_cab_reserve(KERN_cab_t *cab)
{
  KERN_cab_buffer_t *buffer;

  system_lock();
  buffer = cab->free_list;
  if(!buffer)
    {
      /* This is due to the cab being wrongly dimensioned */
      goto out;
    }
  cab->free_list = cab->free_list->next;
 out:
  system_unlock();
  return buffer;
}
```

We make this allocated buffer available to callers of `KERN_cab_get` by setting the CAB's `most_recent` field to be this buffer. The previous "most recent" buffer can be recycled if no job is reading it. All of this is done by `KERN_cab_put`.

Listing 3.5: KERN_cab_put

```
void KERN_cab_put(KERN_cab_t *cab, KERN_cab_buffer_t *buffer
    )
{
    system_lock();
    if(cab->most_recent)
    {
        if(!cab->most_recent->use_counter)
        {
          /*
           * Since no jobs are using the previously "most
               recent" buffer,
           * we free it.
           */
            cab->most_recent->next = cab->free_list;
            cab->free_list = cab->most_recent;
        }
    }
    cab->most_recent = buffer;
    system_unlock();
}
```

The final part of the implementation is actually the first function called for any CAB. `KERN_cab_create` initialises all the memory associated with a CAB. The function is shown in listing 3.6. The function is given a pointer to a block of memory which is large enough to hold all of the buffers and a pointer to one `KERN_cab_t` structure. Each single buffer needs `sizeof(KERN_cab_buffer_t) + buffer_size - 1` bytes (see figure 3.3).

We take the memory argument and make the CAB's free list point to this block. We then initialise this block so that it looks like a linked list of free CAB buffers. The three lines after the for loop take care of the last entry in

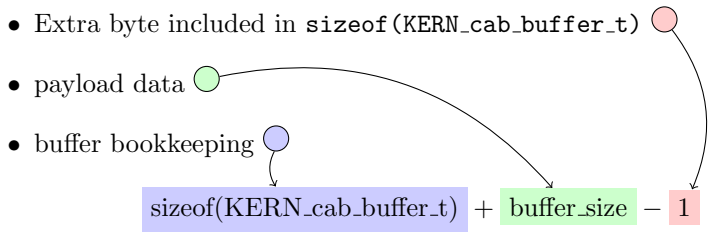


Figure 3.3: CAB buffer dimensioning.

the free list which is a little bit special because its next pointer is to be set to NULL.

Listing 3.6: KERN_cab_create

```
void KERN_cab_create(KERN_cab_t *the_cab, U8 *memory, U32
    number_of_buffers, U32 buffer_size)
{
  KERN_cab_buffer_t *acab;
  U8 *runner;

  the_cab->free_list = (KERN_cab_buffer_t *)memory;
  the_cab->number_of_buffers = number_of_buffers;
  the_cab->most_recent = NULL;

  for(runner=memory;
      runner < memory+(number_of_buffers-1)*(sizeof(
          KERN_cab_buffer_t) + buffer_size - 1);
      runner += sizeof(KERN_cab_buffer_t) + buffer_size - 1)
  {
      acab = (KERN_cab_buffer_t *)runner;
      acab->next = (KERN_cab_buffer_t *)(runner + sizeof(
          KERN_cab_buffer_t) + buffer_size - 1);
      acab->use_counter = 0;
  }
  acab = (KERN_cab_buffer_t *)runner;
  acab->next = NULL;
  acab->use_counter = 0;

}
```

The memory is now in a format suitable for the CAB functions. The only point left to discuss is who performs the CAB initialisation. Just like the jobs and the resources it is the SDL tool which generates the code which calls KERN_cab_create before the scheduler starts. Again, as with the configuration of jobs and resources, a designer doesn't have to use SDL. In such a case, the designer would have to make sure that a CAB was initialised before it was first referenced. The function KERN_cab_create doesn't need to be protected with a system_lock system_unlock mutex. As far as the number of buffers needed to avoid blocking, it is the number of concurrent users of the CAB plus one. To see why, imagine two jobs using a CAB, one is sending and the other is

receiving. The receiver has called `KERN_cab_get` and has one buffer. The writer then sends a message and the buffer it uses becomes the "most recent" buffer. The total number of buffers used is now two. If the writer now wants to send another message then it needs a free buffer returned from `KERN_cab_reserve` which gives us a total of three used buffers. These two jobs will never need more than three buffers if the writer does not have several ongoing calls to `KERN_cab_reserve`.

THE SDL TOOL

The SDL tool was designed to automate the configuration of the kernel. The use of the tool is not compulsory in any way.

A.1 Using SDL

The tool is given two files as input:

- a description of the system's jobs, resources, and CABs.
- a skeleton kernel file.

With this input the SDL tool generates code which when combined with the skeleton kernel file will contain code for the kernel and the initialisation of all the jobs, resources, and CABs. The skeleton kernel file will be the same for each system which uses the same version of the kernel. This file is in fact the C code which has been described in the three steps outlined in this book. The description input file, on the other hand, will most likely be different for each system. This file is called the SDL file.

The input SDL file is read in by the tool and parsed according to the grammar shown in figure A.1.

Figure A.1: SDL Grammar in BNF.

⟨start⟩	→	⟨options⟩ ⟨resources⟩ ⟨cabs⟩ ⟨macros⟩ ⟨jobs⟩
⟨options⟩	→	ϵ | ⟨options list⟩
⟨options list⟩	→	⟨option⟩ | ⟨options list⟩ ⟨option⟩
⟨option⟩	→	"option" ID
⟨resources⟩	→	ϵ | ⟨resources list⟩
⟨resources list⟩	→	⟨resource⟩ | ⟨resources list⟩ ⟨resource⟩
⟨resource⟩	→	"resource" ID "count" NUM "available" NUM
⟨cabs⟩	→	ϵ | ⟨cabs list⟩
⟨cabs list⟩	→	⟨cab⟩ | ⟨cabs list cab⟩
⟨cab⟩	→	"cab" ID "count" NUM
⟨macros⟩	→	ϵ | ⟨macros list⟩
⟨macros list⟩	→	⟨macro⟩ | ⟨macros list macro⟩
⟨macro⟩	→	"macro" ID ⟨use definition list⟩
⟨jobs⟩	→	ϵ | ⟨jobs list⟩
⟨jobs list⟩	→	⟨job⟩ | ⟨jobs list⟩ ⟨job⟩
⟨job⟩	→	"periodic" ID "deadline" NUM "period" NUM "entrypoint" ID | "periodic" ID "deadline" NUM "period" NUM "entrypoint" ID ⟨use definition list⟩ | "sporadic" ID "deadline" NUM "period" NUM "entrypoint" ID | "sporadic" ID "deadline" NUM "period" NUM "entrypoint" ID ⟨use definition list⟩
⟨use definition list⟩	→	⟨use definition⟩ | ⟨use definition list⟩ ⟨use definition⟩
⟨use definition⟩	→	"uses" NUM "of" ID | "uses" ID

An example SDL file is shown in listing A.1. The lines that start with "#" are comments.

Listing A.1: Example SDL file.

```
# -*-sdl-*-
#   app_main.sdl
#

#
# Options
#
# We must define either EDF or DM scheduling.
#
option edf
#option dm

# Another option is to enable HOOKS.
# Uncomment the following line to see what hooks do. Look at
    the jobs.c
# file for the functions that start with the name HOOK_ then
    look at
# kernel.c to see when these functions are called.
#
#option hooks

# resources
#
# We'll need a resource called mutex which has a maximum of
   one unit
# available and when the system starts there will be one
   unit available.
#
resource mutex count 1 available 1

#
# CABs
#
# Here we define a CAB which will be used to send and
   receive integers (32 bit).
# We'll call it temp. The size of an integer is 4 so the
   count will be 4.
cab temp count 4

#
# macros
#
# You can combine CABS and resources into a macro which
   makes the next part
# where we define jobs easier. Have a look at the SDL code
   if you want to see
# what they do. They're just there to make life a little bit
    easier, they are
# not essential.
```

```
#
# jobs
#
# A job can be periodic or sporadic. Each job has a name, a
    deadline, a period,
# and an entrypoint. We can also provide a list of CABs and
    resources that the
# job uses.
#
# j1 is a periodic job to be executed every 100 clock ticks
    with a relative
# deadline of 100 ticks. The C function which does the job's
     work is called
# "job1" (defined in jobs.c). j1 uses 1 unit of the resource
     mutex, and will
# perform at most one CAB send or receive at the same time.

periodic j1 deadline 100 period 100 entrypoint job1 uses 1
    of mutex uses 1 of temp

# j2 is pretty similar to j1.
periodic j2 deadline 200 period 200 entrypoint job2 uses 1
    of mutex uses 1 of temp

# s1 is a sporadic job which is connected to an interrupt.
# The connection is made in the BSP. You can see that this
    job uses no resources
# or CABs.
sporadic s1 deadline 1000 period 1000 entrypoint sporadic1
```

When the example SDL file is processed the output is a C file which is the skeleton kernel file which has a SDL generated prologue and epilogue.

The prologue that would be generated in this case is shown in listing A.2 We can see that this matches the input SDL file and that the definitions here will control the scheduling algorithm and correctly dimension the arrays used to store job and resource descriptions, and the stack used to store system state.

Listing A.2: SDL output, prologue

```
#define OPT_EDF_SCHEDULING
#define MAX_NUMBER_JOBS 3
#define MAX_NUMBER_RESOURCES 1
#define MAX_STACK_DEPTH 6
```

The epilogue is shown in listing A.3. All of the jobs, resources, and CABs are defined as global variables which point to the storage used to hold their configurations. This memory is also configured with the kernel functions `KERN_create_cab`, `KERN_create_resource`, and `KERN_create_job`. The hardware is initialised with the BSP functions and the first job is scheduled with a forced preemption.

Listing A.3: SDL output, epilogue

```
KERN_resource_t *mutex;
U32 mutex_ceiling[2] = {4294967195U, 0U};
extern void job1(void);
KERN_job_t *j1;
```

```
extern void job2(void);
KERN_job_t *j2;
KERN_cab_t temp;
static U8 temp_buffer[38];
extern void sporadic1(void);
KERN_job_t *s1;

void KERN_init(void)
{
        mutex = KERN_resource_create(1, mutex_ceiling);
        j1 = KERN_job_create(100, 100, job1, PERIODIC_JOB);
        j2 = KERN_job_create(200, 200, job2, PERIODIC_JOB);
        KERN_cab_create(&temp, temp_buffer, 2, 4);
        s1 = KERN_job_create(1000, 1000, sporadic1,
            SPORADIC_JOB);
        BSP_init_exception_vectors();
        BSP_init_clock();
        system_unlock();
        job_preempt();
}
```

A.2 Implementation

The SDL tool is created with the help of the compiler tools LEX and YACC. To describe the code which implements the tool would mean doubling the size of this book. There are around 600 lines of code in the kernel and around about 800 lines of code in the SDL tool.

As you can see from the tool output it is a fairly simple translator. The only reasonably interesting part of the tool is the calculation of the resource ceiling tables; even this is quite simple and has been described earlier in the book.

B.1 BSP_deadline_overrun

Name `BSP_deadline_overrun`

Synopsis `void BSP_deadline_overrun(KERN_job_t *j);`

Description This function is called if a job misses its deadline. The argument passed is the value returned from `KERN_job_create` used to create the job. This is a perfect point to try to recover from a timing failure which usually will be a programming error or an overload situation. At every clock tick the kernel will check for missed deadlines. If several jobs miss their deadlines then several calls to this function will be made. This function will be called a maximum of once for a given job during one of its periods.

See also `KERN_job_create`

B.2 BSP_disable_interrupts

Name `BSP_disable_interrupts`

Synopsis `void BSP_disable_interrupts(void);`

Description This function is only called by the kernel's internal function `system_lock` which, along with its sister function `system_unlock`, implement a mutual exclusion mechanism used by the kernel to serialise the execution of kernel code which can alter the kernel's internal data structures. Any interrupts which are completely independent of the kernel and the jobs it schedules are exempt from the functions `BSP_enable_interrupts` and `BSP_disable_interrupts`.

See also `BSP_enable_interrupts`

B.3 BSP_enable_interrupts

Name `BSP_enable_interrupts`

Synopsis `void BSP_enable_interrupts(void);`

Description This function is called by the kernel when it has finished updating its internal data structures. It is called once by the function `KERN_init` as part of the kernel initialisation. All other calls to the function `BSP_enable_interrupts` function are called by the kernel's internal function `system_unlock` which, along with its sister function `system_lock`, implement a mutual exclusion mechanism used by the kernel to serialise the execution of kernel code which can alter the kernel's internal data structures. Any interrupts which are completely independent of the kernel and the jobs it schedules are exempt from the functions `BSP_enable_interrupts` and `BSP_disable_interrupts`.

See also BSP_disable_interrupts, KERN_init

B.4 BSP_init_exception_vectors

Name `BSP_init_exception_vectors`

Synopsis `void BSP_init_exception_vectors(void);`

Description This function is called by the kernel function `KERN_init` to initialise the interrupt controller. The word "interrupt" was not used in the name because it is sometimes useful to have exception vectors initialised so as to catch any programming errors. It is assumed that interrupts which use kernel functions (`KERN_clock_tick`, `KERN_job_arrive`) are disabled and only enabled when the `KERN_init function` calls `BSP_enable_interrupts`.

See also `BSP_enable_interrupts`, `BSP_disable_interrupts`, `KERN_init`

B.5 BSP_init_clock

Name `BSP_init_clock`

Synopsis `void BSP_init_clock(void);`

Description This function is called by `KERN_init` to initialise the real-time clock hardware. It is assumed that the real-time clock will generate interrupts at a fixed frequency. The period between these interrupts is the time base for any period given in calls to the `KERN_create_job` function. The interrupt handler which will be part of the BSP should call the function `KERN_clock_tick` at every interrupt.

See also `KERN_clock_tick`, `KERN_create_job`, `KERN_init`

B.6 HOOK_job_arrive

Name `HOOK_job_arrive`

Synopsis `void HOOK_job_arrive (KERN_job_t *job);`

Description This function is called when a periodic job moves from the inactive state to the pending state, i.e., when the job's period starts. This function will also be called when a sporadic job is moved into the pending state. The function argument is the same value that was returned by the call to `KERN_create_job` when the job was created.

See also `KERN_create_job`

B.7 HOOK_job_finish

Name `HOOK_job_finish`

Synopsis `void HOOK_job_finish (KERN_job_t *job);`

Description This function is called by the kernel when a job finishes execution. The function argument is the same value that was returned by the call to KERN_create_job when the job was created. When this function returns, a preemption is possible.

See also KERN_create_job

B.8 HOOK_job_start

Name HOOK_job_start

Synopsis `void HOOK_job_start (KERN_job_t *job);`

Description This function is called by the kernel just before a job starts execution. The function argument is the same value that was returned by the call to KERN_create_job when the job was created.

See also KERN_create_job

B.9 KERN_cab_buffer_data

Name KERN_cab_buffer_data

Synopsis `void * KERN_cab_buffer_data (KERN_cab_buffer_t *buffer);`

Description Once a buffer has been retrieved from the CAB with a call to KERN_cab_get, the actual payload data can be gotten at with a call to KERN_cab_buffer_data. The following example code shows how this is done.

```
if(!(buffer = KERN_cab_get(&temp)))
  {
    return;
  }
data = (U32*)KERN_cab_buffer_data(buffer);
```

See also KERN_cab_get

B.10 KERN_cab_create

Name KERN_cab_create

Synopsis `void KERN_cab_create(KERN_cab_t *the_cab, U8 *memory, U32 number_of_buffers, U32 buffer_size);`

Description Before any communication can be performed with a CAB, it must be initialised. The parameters are used for the following:

`the_cab` This is the CAB we will initialise.

`memory` This is a pointer to a memory area which will hold all of the CAB's buffers. The size of this area in bytes can be calculated as follows:

$$number\_of\_buffers * (sizeof(KERN\_cab\_buffer\_t) + buffer\_size - 1)$$

`number_of_buffers` The number of buffers we need is equal to the number of concurrent read and writes that can be ongoing at any one time plus one.

`buffer_size` The size of each buffer in bytes.

See also `KERN_cab_get`, `KERN_cab_reserve`

B.11 KERN_cab_get

Name `KERN_cab_get`

Synopsis `KERN_cab_buffer_t * KERN_cab_get(KERN_cab_t *cab);`

Description If we want to read a buffer from a CAB then we need to call `KERN_cab_get`. This will return the most recently written to buffer. When we're finished we need to call `KERN_cab_unget`.

```
if(!(buffer = KERN_cab_get(&temp)))
  {
    return;
  }
...
KERN_cab_unget(&temp, buffer);
```

See also `KERN_cab_unget`

B.12 KERN_cab_put

Name `KERN_cab_put`

Synopsis `void KERN_cab_put(KERN_cab_t *cab,`
`KERN_cab_buffer_t *buffer);`

Description We call `KERN_cab_put` to indicate to the CAB that the buffer is to be made the least recently written buffer. This means that all jobs which call `KERN_cab_get` will have the buffer returned to them until the next call to `KERN_cab_put`.

```
buffer = KERN_cab_reserve(&temp);
data = (U32 *)KERN_cab_buffer_data(buffer);
*data = long_computation();
KERN_cab_put(&temp, buffer);
```

See also `KERN_cab_reserve`

B.13 KERN_cab_reserve

Name `KERN_cab_reserve`

Synopsis `KERN_cab_buffer_t * KERN_cab_reserve(KERN_cab_t *cab);`

Description We call KERN_cab_reserve when we want to get a free buffer from a CAB.

```
buffer = KERN_cab_reserve(&temp);
data = (U32 *)KERN_cab_buffer_data(buffer);
*data = long_computation();
KERN_cab_put(&temp, buffer);
```

See also KERN_cab_put

B.14 KERN_cab_unget

Name KERN_cab_unget

Synopsis void KERN_cab_unget(KERN_cab_t *cab, KERN_cab_buffer_t *buffer);

Description When we've completed processing a CAB buffer we indicate to the CAB that this buffer may be a candidate for recycling. We can do this by calling KERN_cab_unget.

```
if(!(buffer = KERN_cab_get(&temp)))
{
  return;
}
...
KERN_cab_unget(&temp, buffer);
```

See also KERN_cab_get

B.15 KERN_clock_tick

Name KERN_clock_tick

Synopsis void KERN_clock_tick (void);

Description This function should be called by the real-time clock interrupt service routine which is defined by the BSP. Calling this function may possibly lead to a preemption as the function will check all inactive jobs to see if their period has started. This function will also check for deadline overruns so the function BSP_deadline_overrun may be called.

See also None.

B.16 KERN_init

Name KERN_init

Synopsis void KERN_init(void);

Description The BSP should call this function once the machine has booted and the stack is configured. This function starts by calling the `KERN_job_create` function once for each job defined. The function then calls the BSP function `BSP_init_exception_vectors` then `BSP_init_clock`. The system is unlocked with a call to `BSP_enable_interrupts`, then the highest priority job is started.

See also None.

B.17 KERN_job_arrive

Name `KERN_job_arrive`

Synopsis `void KERN_job_arrive (KERN_job_t *job);`

Description This function is called to release a sporadic job. This function is reentrant and should be called with interrupts enabled. The argument passed to the function is the return value from the call to `KERN_job_create` used to create the sporadic job.

See also `KERN_job_create`

B.18 KERN_job_create

Name `KERN_job_create`

Synopsis `KERN_job_t *KERN_job_create (U32 period, U32 relative_deadline, void (*function)(void), KERN_job_type_t type);`

Description This function is used to create periodic and sporadic jobs. The `period` and `relative_deadline` arguments are multiples of the BSP's real-time clock interrupt period. The `function` argument is the job's entrypoint. The `type` argument is either `PERIODIC_JOB` or `SPORADIC_JOB`. The return value of this function is used as an argument to the `HOOK_` functions or passed as an argument to the `KERN_job_arrive` and `BSP_deadline_overrun` functions.

See also `BSP_deadline_overrun`, `HOOK_job_arrive`, `HOOK_job_finish`, `HOOK_job_start`, `KERN_job_arrive`

B.19 KERN_resource_create

Name `KERN_resource_create`

Synopsis `KERN_resource_t *KERN_resource_create (U32 available, U32 *ceiling);`

Description Before any resource is used with `KERN_resource_request` and `KERN_resource_release`, it must be created with `KERN_resource_create`. The argument `available` tells the system the maximum number of units of this resource will be available at any given time. The `ceiling` argument is an array of unsigned 32-bit numbers which give the resource

ceiling for a given number of units being available for the resource. If `available` was 17, then the `ceiling` array would need to have 18 elements — the extra element is for zero meaning no units left. An example use of resources would be as a mutex. In this case the `available` argument will be 1 so that only one job request can be outstanding at any given time. The `ceiling` array would have a first element so high, all other jobs which would want to use the mutex would block. The next element in the array would be so low as to not block any job which wanted access to the mutex. Example:

```
U32 mutex_ceiling[2] = {4294967195U, 0U};
...
mutex = KERN_resource_create(1, mutex_ceiling);
```

See also `KERN_resource_release`, `KERN_resource_request`

B.20 KERN_resource_release

Name `KERN_resource_release`

Synopsis `void KERN_resource_release (void);`

Description This function is used to release the resources which were requested with the calling job's closest previous call of the function `KERN_resource_request`. Since the order of resource release is the opposite of resource request there is no need for the system designer to identify which resource will be released. The kernel can keep all the information it needs to match releases to the correct requests with a stack data structure. The following function is used to safely swap the contents of two shared variables `a` and `b`:

```
void job (void *dummy)
{
  int temp;
  KERN_resource_request(a_mutex, 1);
  KERN_resource_request(b_mutex, 1);
  temp = a;
  a = b;
  b = temp;
  KERN_resource_release(); /* releases b_mutex */
  KERN_resource_release(); /* releases a_mutex */
}
```

It is important to remember that all resources which are requested during one activation period must be released during the same period. As a consequence of this call, the system ceiling may be lowered which will mean the current job may be preempted.

See also `KERN_resource_request`

B.21 KERN_resource_request

Name KERN_resource_request

Synopsis `void KERN_resource_request (KERN_resource_t *res, U32 amount);`

Description This function is called to request `amount` units of the resource `res`. With the Stack Resource Policy there is a guarantee that the calling job will not block. As a side effect of this function call, the system ceiling may be raised. Only jobs with a preemption level higher than the system ceiling can preempt the job which called this function.

See also KERN_resource_release

NOMENCLATURE

$\mu_R(J)$	The number of units of resource R that the job J will request.
ν_R	Current amount of available units for resource R.
$\overline{\pi}$	The maximum of all the current ceilings of all the resources.
$\overline{R}_{\nu_R}$	The preemption ceiling for resource R which currently has ν_R units available.
$\pi(J)$	The preemption level for job J.
C	Job execution time.
D	Relative deadline.
J	A job.
N_R	Maximum number of units for a resource R.
R	A resource.
T	Job period.

BIBLIOGRAPHY

[Bak91] T. P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 1991.

[But97] Giorgio C. Buttazzo. *Hard Real-time Computing Systems.* Kluwer, 1997.

[CL73] J.W. Layland C.L. Liu. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery, Vol. 20, No.1, pp. 46-61*, January 1973.

[Cla89] D. Clark. Hic: An Operating System for Hierarchies of Servo Loops. *Proceedings of the IEEE International Conference on Robotics and Automation*, 1989.

[Gal95] Bill O. Gallmeister. *POSIX.4: Programming for the Real World.* O'Reilly and Associates, 1995.

[JL82] J.W. Whitehead J. Leung. On the Complexity of Fixed Priority Scheduling of Periodic Real-time Tasks. *Performance Evaluation 2(4)*, July 1982.

[KJ93] D.L. Stone K. Jeffay. Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. *Proceedings of the IEEE Real-time System Symposium, pp. 212-221*, December 1993.

[SZ89] H. Kopetz A. Damm C. Koza M. Mulazzani W. Schwabla C. Senft and Z. Zainlinger. Distributed Fault-tolerant Real-time Systems; The MARS Approach. *IEEE Micro 9(1)*, February 1989.

INDEX