Real-Time Schedulability Tests for Preemptive Multitasking

C. J. Fidge[*]

Abstract

When developing multitasking real-time systems, schedulability tests are used to formally prove that a given task set will meet its deadlines. A wide range of such tests have appeared in the literature. This tutorial acts as a guide to the major tests available for preemptive multitasking applications.

1 Introduction

Safety-critical applications often involve several distinct activities, each of which has ‘hard’ inviolable timing constraints (Burns and Wellings, 1990). Developing such systems so as to guarantee that no critical task ever misses a deadline is a significant intellectual challenge.

In the past programmers resorted to a rigid, pre-determined order for execution of (parts of) tasks, so that the times at which jobs will complete could be predicted in advance. Unfortunately this *cyclic executive* model results in programs that are hard to understand and maintain because the code for logically independent tasks is interleaved. Such methods are now being superseded by *preemptive scheduling* approaches (Locke, 1992).

However, because preemptive scheduling results in an order for task execution that changes dynamically, it is difficult to know in advance whether each task will meet its deadlines or not. A formal tool to overcome this is the notion of a *schedulability test*. Given information such as the execution time of each task, the frequency with which it needs to run, and the particular scheduling policy used, these tests tell the programmer whether a given task set can meet its deadlines.

Over the years a bewildering variety of these tests have been proposed, for a wide range of scheduling policies. They vary considerably in their complexity and capabilities. This review summarises the most well-known tests

[*]Software Verification Research Centre, School of Information Technology, The University of Queensland, Queensland 4072, Australia. Email: `cjf@it.uq.edu.au`

currently available, using consistent notations and terminology, in order to provide a comparative catalogue of the tests available. The survey considers tests for *hard real-time* tasks, executing under a *preemptive* scheduling policy, in a *uniprocessor* environment.

The tests work by considering a worst-case scenario and checking that each task gets a sufficient allocation of shared resources in this situation; if so then it will also succeed under more favourable circumstances. Historically, there have been two distinct approaches. Tests based on the notion of *processor utilisation*, i.e., the percentage of processor time that may be occupied by tasks, developed first. Later work saw the emergence of tests based on *response times*, i.e., the exact duration by which tasks may be delayed. Examples of both methods are considered here. Ultimately, however, the two approaches have been recognised as equivalent (Lehoczky, 1990, p.203) (Audsley et al., 1995, p.182), offering the same capabilities in different styles.

As far as possible the tests are grouped into broad categories based on their capabilities. (This grouping is inevitably somewhat arbitrary due to the varying mix of features supported.)

This report is *not* a self-contained tutorial on schedulability theory; complementary tutorials by Audsley et al. (1995) and Mercer (1992) give gentler introductions to the field.

Section 2 briefly reviews the overall computational model assumed by scheduling theory. Section 3 describes schedulability tests developed for sets of independent, non-communicating tasks. Section 4 describes tests for interacting, communicating task sets. Section 5 presents a tabular summary of the capabilities of each test reviewed. Precise citations are given throughout so that the reader may easily access further information on any of the tests presented, especially their proofs of correctness and examples of their use. A glossary of terminology is provided as an appendix, again accompanied by precise citations throughout. **Bold** text denotes a cross reference to the glossary.

2 Computational model for preemptive multitasking

Hard real-time schedulability theory uses a simplified, abstract computational model to represent the behaviour of a preemptive, multitasking system (Audsley et al., 1993, §2).

The model assumes that the programmer wishes to implement a **task set** on a particular processor. Each task i **arrives** infinitely often, each arrival separated from the last by at least T_i time units. A **periodic** task arrives regularly with a separation of *exactly* T_i time units. A **sporadic** task arrives irregularly with each arrival separated from its predecessor by *at least* T_i time units. (**Aperiodic** tasks, which arrive irregularly with no minimum separation, are not usually considered because hard real-time guarantees cannot be made for them.)

At each arrival, task i issues a notional **invocation request** for up to C_i units of processor time, its worst case **computation time**. Each invocation of task i must have this request satisfied before its **deadline** D_i expires.

The scheduler services these requests according to a particular *scheduling policy*. Each task making a request is notionally placed in a **ready queue** (ISO, 1994, §D.2.1) at which time the task is said to be **released** (Audsley et al., 1993, §2). The scheduler selects a task to run from the ready queue as per the scheduling policy it implements. Tasks of higher priority can **preempt** the running task i, resulting in a degree of **interference** I_i to the progress of task i. Tasks stop being ready by **suspend**ing themselves (by executing a 'delaying' statement). Suspended tasks wait in a separate **delay queue** until they become ready again (Burns et al., 1995, p.476).

Scheduling decisions are based on the **priority** of ready tasks. In *static-priority* scheduling there is a fixed **base priority** associated with each task, although the task may temporarily acquire a higher **active priority** at run time. Static-priority policies include **rate monotonic** and **deadline monotonic** scheduling. When discussing static-priority policies we assume there are n tasks, with (usually) *unique* base priorities, ranked from highest-priority task 1 to lowest-priority task n.

Dynamic-priority scheduling policies assign variable priorities at run time. Dynamic-priority policies include **earliest deadline first** and **least laxity** scheduling. Dynamic-priority policies can lead to better processor utilisation than static-priority ones, but are harder to implement (Manabe and Aoyagi, 1995, p.213), and thus less common.

All communication between tasks residing on the same processor is through controlled access to shared variables, with mutual exclusion guaranteed by semaphores or a similar *locking protocol*. This communication model supports timing predictability because the worst-case **blocking** time B_i that task i may experience while awaiting access to a shared variable can be determined *a priori*. Locking protocols include the **priority ceiling**, **ceiling locking**, **kernelised monitor** and **stack resource** protocols.

For simplicity the computational model usually assumes that only tasks consume time. The scheduler itself is treated as if it executes instantaneously. The overheads of context switching, interrupt handling, and shared resource locking are factored into the worst case computation time for each task invocation (Klein and Ralya, 1990, p.12) (Leung and Whitehead, 1982, p.238) (Audsley et al., 1994).

C_i Worst case **computation time** that may be required by an invocation of task i.

T_i Lower bound between successive **arrival**s of task i. This is the period of a periodic task and the minimum separation of the triggering event for a sporadic task.

D_i The **deadline** for each invocation of task i, measured from its arrival time (Tindell et al., 1994, p.149).[a] Usually $D_i \leq T_i$ (Audsley and Burns, 1990, p.6) (Tindell et al., 1994, p.134). If no deadline is specified for a periodic task i then $D_i = T_i$ is assumed.

[a] Audsley et al. (1993, p.285) measure D_i from *release* time.

Figure 1: Task characteristics specified by the programmer (Audsley et al., 1993, §3).

3 Independent tasks

The tests in this section apply to a set of n tasks, residing on the same processor, when the tasks are independent (i.e., they do not interact or otherwise communicate using shared resources and hence cannot block one another). Mathematical symbols used are defined in Figures 1 and 2.

3.1 Deadlines equal periods

The following tests apply to sets of independent **periodic** tasks whose deadlines are equal to their periods.

Rate monotonic scheduling In a seminal paper, Liu and Layland (1973) proved that any such task set is schedulable using **rate monotonic** scheduling if

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

Expression C_i/T_i is the percentage of time that task i will occupy the processor at each arrival, in the worst case. The test thus states that the sum of these **processor utilisation** percentages, for all n tasks, must not exceed the 'utilisation bound' (Liu and Layland, 1973, §5) (Baker, 1991, p.68). The utilisation bound expression $n(2^{1/n} - 1)$ converges to 69% for large values of n, so task sets passing this test may have low processor utilisation. This is a **sufficient** but not **necessary** test (Audsley et al., 1993, p.284).

B_i Worst case **blocking** time an invocation of task i may experience due to shared resources being locked by lower-priority tasks.

I_i Worst case **interference** an invocation of task i may experience due to preemptions by higher-priority tasks.

U_i Worst case **processor utilisation** percentage by task i and tasks of higher priority.

J_i Worst case **release jitter** for an invocation of task i due to scheduling delays.

R_i Worst case **response time** for an invocation of task i, measured from its *arrival* time.[a] Usually $R_i = J_i+C_i+B_i+I_i$. A schedulable task must have $R_i \leq D_i$.

r_i Worst case **response time** for an invocation of task i, measured from its *release* time (Audsley et al., 1993, p.286). Usually $r_i = C_i + B_i + I_i$. When there is no release jitter $r_i = R_i$.

[a]Tindell et al. (1994, p.150) use "r_i" where we use "R_i".

Figure 2: Task characteristics defined by the scheduling policy and locking protocol used (Audsley et al., 1993, §3).

Lehoczky et al. (1989) then developed a more discerning test. Let

$$U_i(t) = \frac{\sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil}{t} ,$$

be the worst-case processor utilisation needed by all tasks j of priority greater than or equal to i during an interval of duration t (Lehoczky et al., 1989, p.167). The amount of processor time required is calculated as the sum of each computation time C_j multiplied by the number of periods $\lceil t/T_j \rceil$ started by task j within t time units.[1] This is divided by t to give the utilisation percentage. The test then states that the task set is schedulable if,

(2) $$\text{for all tasks } i, \quad \min_{0<t\leq T_i} U_i(t) \leq 1 .$$

In other words, for each task i, at every moment of time t within its period T_i, processor utilisation U_i up until time t must not exceed 100%. The equation

[1]For some number x, $\lceil x \rceil$ is the smallest integer greater than or equal to x.

defining U_i assumes the worst-case scenario where all tasks arrive simultaneously, so it is only necessary to test that each task i will meet its *first* deadline to know that it will *never* miss a deadline (Lehoczky et al., 1989, p.167).

Although harder to check, schedulability test 2 may pass task sets that test 1 fails, and thus allows higher processor utilisation. Test 2 is both necessary and sufficient (Lehoczky et al., 1989, p.168).

A variant, also defined by Lehoczky et al. (1989), shows that rather than checking *all* times t it is possible to reduce the number of calculations required for test 2 by checking only certain *scheduling points*. These are all multiples of the periods of each higher-priority task. For some task i let

$$P_i = \{kT_j \mid 1 \leq j \leq i, \quad k = 1, \ldots, \left\lfloor \tfrac{T_i}{T_j} \right\rfloor\}$$

be the scheduling points that occur within its period T_i, i.e., each absolute time of the k^{th} arrival of each higher or equal priority task j (Lehoczky et al., 1989, p.167). Task j can arrive up to $\lfloor T_i/T_j \rfloor$ times in T_i time units.[2] A set of independent periodic tasks using rate monotonic scheduling will then meet all its deadlines if (Lehoczky et al., 1989, p.168),

$$\text{(3)} \qquad \text{for all tasks } i, \quad \min_{t \in P_i} U_i(t) \leq 1 \ ,$$

where utilisation $U_i(t)$ is as defined for test 2 above.

Manabe and Aoyagi (1995, §4) recently suggested a further optimisation of test 3. It defines the significant scheduling points for task i as only T_i, the largest multiple of T_{i-1} less than or equal this point, the largest multiple of T_{i-2} less than or equal this new point, and so on.

Earliest deadline first scheduling Since the dynamic **earliest deadline first** scheduling policy is **optimal** for independent tasks, Liu and Layland (1973, p.186) showed that any task set is schedulable under this policy if processor utilisation by all tasks does not exceed 100% (Giering III and Baker, 1994, p.55). A sufficient and necessary test (Liu and Layland, 1973, p.183) for independent tasks using earliest deadline first scheduling is thus trivial (Liu and Layland, 1973, p.184) (Baker, 1991, p.69):

$$\text{(4)} \qquad \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \ .$$

3.2 Deadlines less than periods and sporadic tasks

The following tests were developed to relax the assumptions that all tasks are periodic and have deadlines equal to their periods. They allow **sporadic** tasks

[2]For some number x, $\lfloor x \rfloor$ is the largest integer less than or equal to x.

to be introduced to the task set, and deadlines to be less than or equal to task interarrival times.

Deadline monotonic scheduling Audsley et al. (1991, §2) observed that an invocation of some task i, in a task set with static priorities allocated according to the **deadline monotonic** scheduling policy, will experience worst-case interference

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j$$

from tasks with higher deadline-monotonic priorities. I_i is the sum, for each higher-priority task j, of its computation time C_j multiplied by the number of times $\lceil D_i/T_j \rceil$ it arrives before the deadline D_i of task i.

The entire deadline-monotonic task set can be scheduled if,

$$\text{(5)} \qquad \text{for all tasks } i, \quad \frac{C_i}{D_i} + \frac{I_i}{D_i} \le 1 \ .$$

Since deadlines can be less than their periods, D_i is used instead of T_i to determine utilisation. The test checks that the processor utilisation C_i/D_i by task i before its deadline, plus the degree of interference I_i/D_i it experiences before the deadline, does not exceed 100%. This test is suitable for sporadic as well as periodic tasks (no explicit **sporadic server** is needed to handle sporadic events).

The above interference measure is overly pessimistic, however, because it includes the entire worst-case computation time C_j of each task invocation that *arrives* even though part of task j may complete *after* deadline D_i has passed. A more precise definition is (Audsley et al., 1991, §2)

$$I_i = \sum_{j=1}^{i-1} \left(\left\lfloor \frac{D_i}{T_j} \right\rfloor C_j + \min \left(C_j, D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j \right) \right) \ .$$

For each higher priority task j, the left-hand term defines the overhead of whole invocations of task j completed before D_i and the right-hand term defines whatever fraction of C_j can be accommodated in the time remaining before deadline D_i.

Both variants of test 5 are sufficient but not necessary (Audsley et al., 1991, §2) because they assume a worst-case interleaving of higher-priority tasks that may never actually occur. A necessary test for deadline monotonic scheduling must account for the actual interleaving of tasks; the tests presented in Section 4.3 below do this. (Audsley et al. (1991, §2) also present an algorithm that achieves such a result.)

Recently, Manabe and Aoyagi (1995, §5) proposed a test for deadline monotonic scheduling, based on the principles illustrated in test 3. Again the test works by checking processor utilisation for task i, and higher-priority tasks, at all its schedulability points (Manabe and Aoyagi, 1995, p.217):

(6) $$\text{for all tasks } i, \quad \min_{t \in Q_i} U_i(t) \leq 1 \ .$$

Term $U_i(t)$ is defined as for test 2 above. The new feature is the choice of scheduling points,

$$Q_i = \{D_i\} \cup \{kT_j \mid 1 \leq j \leq i-1, \quad k = 1, \ldots, \left\lfloor \tfrac{D_i}{T_j} \right\rfloor\} \ .$$

Here the significant scheduling points for an invocation of task i are its deadline D_i, and all multiples k of the period of higher priority tasks j that can arrive within D_i time units. (As they did for test 3, Manabe and Aoyagi (1995, Def.6) also suggest an optimisation of this definition to further reduce the number of scheduling points that need to be evaluated.)

Static-priority scheduling The following tests apply to any static assignment of base priorities to tasks, including that defined by the rate monotonic and deadline monotonic scheduling policies.

Recently, Park et al. (1996) proposed an approach to schedulability testing based on linear programming. Unlike the other tests we consider, it can be used when worst-case computation times are not yet known. For each task i, their minimum utilisation bound M_i is defined as the result of applying the following procedure (Park et al., 1996, p.59):

$$\begin{aligned} \text{Find the smallest} \quad & M_i = \sum_{j=1}^{i} \frac{C_j}{T_j} \\ \text{subject to} \quad & \sum_{j=1}^{i} C_j \left\lceil \frac{D_i}{T_j} \right\rceil = D_i \ , \end{aligned}$$

where each C_j must be greater than 0. In other words, we must find suitable computation times for all tasks j, of equal or higher priority than i, that minimise the processor utilisation bound for task i. The condition on the second line ensures that all task arrivals by some task j, before the deadline D_i of task i, are completed at exactly the deadline. (The *minimum* utilisation bound is found when there is no processor idling before the deadline (Park et al., 1996, p.59).) Note that this allows us to *discover* suitable computation times, in terms of the task deadlines and periods, rather than supplying them.

An entire task set is then schedulable if,

(7) $$\text{for all tasks } i, \quad \sum_{j=1}^{n} \frac{C_j}{T_j} \leq M_i \ .$$

In other words, the overall processor utilisation for all n tasks must be less than every individual task utilisation bound M_i (Park et al., 1996, p.60).

The following test, unlike its predecessors, is not based on the principle of processor utilisation bounds. Joseph and Pandya (1986) showed that a task set will meet all its deadlines if,

$$(8) \qquad \begin{aligned} \text{for all tasks } i, \quad & R_i \leq D_i \ , \\ \text{where} \quad & R_i = C_i + I_i \\ \text{and} \quad & I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \ . \end{aligned}$$

In other words, for each task i, the test checks that its response time R_i is always less than its deadline D_i. The worst case response time for an invocation of task i is defined to be its worst case computation time C_i plus its worst case interference I_i.

Interference term I_i determines how much preemption task i will experience, due to higher-priority tasks j, during the interval of time defined by R_i. For each task j this interference is execution time C_j multiplied by the number of arrivals $\lceil R_i/T_j \rceil$ that j may have in R_i time units. Thus interference during R_i is defined in terms of the number of preemptions that may occur during the interval of time defined by R_i: the definition of R_i is recursive! Joseph and Pandya observe that "equations of this form do not lend themselves easily to analytical solution" (Joseph and Pandya, 1986, p.391), but see Section 4.3 below.

4 Communicating tasks

The tests above assumed that tasks do not interact. The tests in this section allow for communication between tasks on the same processor, via access to shared variables, controlled using a particular locking protocol.

4.1 Deadlines equal periods

The following tests apply to communicating periodic (or sporadic with a periodic server) tasks whose deadlines equal their period.

Rate monotonic scheduling Sha et al. (1987) showed that any such task set, when using the **priority ceiling protocol**, is schedulable if the following generalisation of test 1 holds (Baker, 1991, p.68):

$$(9) \qquad \text{for all tasks } i, \quad \sum_{j=1}^{i} \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq i(2^{\frac{1}{i}} - 1) \ .$$

Thus, for every task i, the sum of processor utilisation by equal or higher priority tasks j, plus processor utilisation lost due to blocking, must be less than the bound. Blocking figure B_i is the worst-case computation time of the longest critical section of a task of *lower* priority than task i (Chen and Lin, 1990, p.329) (Baker, 1991, p.69). Expression B_i/T_i is thus the worst case 'blocking utilisation' for task i: the task can be blocked at most once in period T_i when the priority ceiling protocol is enforced.

An equivalent variant of test 9, applicable to the full set of n tasks, is (Sha et al., 1989, p.248)

$$(10) \qquad \frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} + \max\left(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \leq \; n(2^{\frac{1}{n}} - 1) \; .$$

Test 9 also applies when sporadic tasks are introduced using the **sporadic server algorithm** (Sprunt et al., 1989, p.57).

A further generalisation of test 9 accounts for tasks whose priority does not obey rate monotonicity (Gomaa, 1993, §11.4.8). Since the minimum arrival separation of a sporadic task, such as an interrupt server, may not necessarily be a measure of its true importance or required responsiveness, the test allows for the base priority of sporadic tasks to be higher than the value that would be allocated according to their interarrival times. (Under these circumstances tests 1 and 9 are not sufficient to guarantee schedulability (Gomaa, 1993, p.133).)

For some periodic task i let H_i be the set of tasks with higher priority than, and periods shorter than, task i. These tasks obey the usual rate monotonicity relationship. Let S_i be those tasks with higher priority than, but periods *longer* than, task i. These tasks do *not* obey rate monotonic priority allocation. Preemption by such a task is treated in the test like blocking: like lower-priority tasks they can block task i at most once during period T_i because their period is longer than T_i. The schedulability test, assuming use of the priority ceiling protocol, is then

$$(11) \quad \text{for all tasks } i, \quad \sum_{j \in H_i} \frac{C_j}{T_j} + \frac{C_i}{T_i} + \frac{B_i}{T_i} + \sum_{k \in S_i} \frac{C_k}{T_i} \leq n(2^{\frac{1}{n}} - 1) \; .$$

For each task i the first term is the degree of interference i can experience in its period due to tasks j in H_i: each task j may preempt an invocation of task i several times because $T_j \leq T_i$. The second term is the processor utilisation of task i itself. The third term is the degree of blocking i may encounter from lower (rate monotonic) priority tasks. The fourth term is the preemption utilisation that i can experience due to tasks k in S_i: each task k may preempt an invocation of task i at most once because $T_k \geq T_i$, so T_i is used instead of T_k to calculate utilisation lost due to these tasks.

Again this test is not a necessary one and a generalisation of test 3 is also available that may pass task sets that test 11 fails. Sprunt et al. (1989, p.57) use the following test in the situation where rate monotonic scheduling is used for periodic tasks except for a high-priority sporadic server task whose assigned priority may be higher than that warranted by its minimum interarrival time:

$$\text{for all tasks } i, \quad \min_{t \in P_i} \left(U_i(t) + \frac{B_i}{t} \right) \leq 1 \; . \tag{12}$$

P_i is defined as for test 3, and $U_i(t)$ as for test 2 above. At each scheduling point t the first term defines utilisation due to periodic tasks with rate-monotonic priorities higher or equal to i, and the second is the percentage of blocking due to tasks with lower rate-monotonic priorities. However, when calculating the blocking time B_i, for a task i that has a rate monotonic priority higher than the *rate monotonic* priority of a sporadic server, then the execution time of that server must be counted as a possible source of 'blocking' (Sprunt et al., 1989, p.58). Such high-priority servers may 'block' (in fact preempt!) task i (once) during T_i.

Earliest deadline first scheduling Test 4 can be trivially extended for systems using earliest deadline first scheduling and the **kernelised monitor protocol** (Chen and Lin, 1990, p.328):

$$\sum_{i=1}^{n} \frac{C_i + B}{T_i} \leq 1 \; . \tag{13}$$

Here blocking time B is the longest computation time that *any* task may spend in *any* critical section. If B is large then task sets passing this test will have low utilisation.

Chen and Lin (1990) then showed that a much tighter bound can be achieved when their dynamic version of the priority ceiling protocol is used instead of the kernelised monitor (Giering III and Baker, 1994, p.56) (Baker, 1991, p.69):

$$\sum_{i=1}^{n} \left(\frac{C_i}{T_i} + \frac{B_i}{T_i} \right) \leq 1 \; . \tag{14}$$

Blocking time in test 14 is again due to lower-priority tasks only, thus showing that this scheduling approach is more efficient than that assumed by test 13.

4.2 Deadlines less than periods

The following test applies to communicating periodic tasks, and allows their deadlines to be less than or equal to their periods.

Earliest deadline first scheduling Baker (1991) developed a tighter bound on utilisation than in test 14 by using the **stack resource protocol** (Baker, 1990) to control semaphore locking, under earliest deadline first scheduling. A set of tasks, with higher priorities allocated to tasks with shorter (static!) deadlines D_i (Baker, 1991, §2.5.1), is then schedulable if (Baker, 1991, p.83),

$$\text{(15)} \qquad \text{for all tasks } i, \quad \sum_{j=1}^{i} \frac{C_j}{D_j} + \frac{B_i}{D_i} \leq 1 \ .$$

For each task i, B_i is the execution time of the longest critical section in some task k such that $D_i < D_k$. However even when task deadlines equal periods this test gives a better bound than test 14 (Baker, 1991, p.84) because the stack resource model offers better schedulability.

4.3 Deadlines less than periods and sporadic tasks

The following tests apply to communicating periodic, sporadic and, in later cases, **sporadically periodic** tasks, with deadlines that may be less than or equal to their interarrival times.

Static-priority scheduling Audsley et al. (1993) define a general static-priority schedulability test for communicating tasks by extending test 8. It is suitable for rate monotonic and deadline monotonic priority allocations. The test applies to truly sporadic tasks as well as periodic ones.

A set of periodic and sporadic tasks is schedulable if,

$$\text{(16)} \qquad \begin{aligned} \text{for all tasks } i, \quad & R_i \leq D_i \ , \\ \text{where} \quad & R_i = C_i + B_i + I_i \\ \text{and} \quad & I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \ . \end{aligned}$$

As before, the test checks that the response time R_i of task i is always less than its deadline D_i. Assuming use of the priority ceiling protocol, blocking time B_i equals the longest critical section of any lower priority task accessing a shared variable with a ceiling priority as great as the priority of task i (Audsley et al., 1993, p.286).

Again the definition of response time is recursive. Fortunately, however, Audsley et al. (1993, p.287) show that it is possible to solve this equation iteratively. Let R_i^x be the x^{th} approximation to the value of R_i. Starting with $R_i^0 = 0$, equation

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

converges to R_i. Evaluation stops either when the equation has converged, i.e., $R_i^{x+1} = R_i^x$, or, because $R_i^{x+1} \geq R_i^x$ for any x, iteration can stop as soon as $R_i^{x+1} > D_i$ in which case the test has failed.

In cases where the scheduler implementation may introduce appreciable **release jitter** J_i for task i, test 16 can be generalised to (Audsley et al., 1993, p.288),[3]

(17)
$$\begin{aligned} \text{for all tasks } i, \quad & R_i \leq D_i \ , \\ \text{where} \quad & R_i = J_i + r_i \\ \text{and} \quad & r_i = C_i + B_i + I_i \\ \text{and} \quad & I_i = \sum_{j=1}^{i-1} \left\lceil \frac{r_i + J_j}{T_j} \right\rceil C_j \ . \end{aligned}$$

Interval r_i is used instead of R_i to calculate interference because the task cannot be preempted before it is released. Also, whereas the worst-case scenario is usually a moment when all tasks arrive simultaneously, this is not the case when there may be release jitter. The worst case for some task i in this situation is when it is *released* at the same moment that all higher-priority tasks j are themselves released (Audsley et al., 1993, p.288). The jitter value J_j accounts for this by allowing a preemptive invocation of task j to arrive *before* task i is released. Again the recursive definition can be solved iteratively.

Audsley et al. (1993) then extend these tests to allow for **sporadically periodic** tasks. Figure 3 shows the symbols used in this case. A sporadically-periodic task j has sporadic 'outer' arrivals separated by T_j but, having arrived, then rearrives repeatedly according to some 'inner' period t_j a fixed number of times m_j (Audsley et al., 1993, §5). This behaviour is found in "bursty" communications media, for instance. To use the inner period as the task's interarrival time would be unnecessarily pessimistic and lead to low processor utilisation, so a more complex analysis is desirable.

The approach is to extend the definition of I_i from test 16 to account for preemptions by a higher-priority sporadically-periodic task j (Audsley et al., 1993, p.290):

(18)
$$\begin{aligned} \text{for all tasks } i, \quad & R_i \leq D_i \ , \\ \text{where} \quad & R_i = C_i + B_i + I_i \\ \text{and} \quad & I_i = \sum_{j=1}^{i-1} \left(\min \left(\left\lceil \frac{R_i - T_j \left\lfloor \frac{R_i}{T_j} \right\rfloor}{t_j} \right\rceil , m_j \right) + m_j \left\lfloor \frac{R_i}{T_j} \right\rfloor \right) C_j \ . \end{aligned}$$

[3] We use a different definition of D_i than Audsley et al. (1993, p.285), starting from *arrival* instead of release time.

T_j Lower bound between successive arrivals of **sporadically periodic** task j's outer 'period'.

t_j The inner period of sporadically periodic task j.

m_j The number of inner arrivals of sporadically periodic task j per each outer arrival: $m_j t_j \leq T_j$.

Figure 3: Task characteristics specified for a sporadically periodic task (Audsley et al., 1993, p.290).

As before, interference I_i, due to each higher priority task j, is determined by the number of arrivals of j multiplied by its execution time C_j, in the interval R_i. The second major term in the summation determines the total number of arrivals occurring due to 'complete' outer arrivals in R_i, i.e., the number of outer arrivals $\lfloor R_i/T_j \rfloor$ times the number of inner arrivals m_j per outer arrival. The first major term is the number of inner arrivals due to a remaining 'incomplete' outer arrival. This is found by subtracting the time consumed by complete outer arrivals $T_j \lfloor R_i/T_j \rfloor$ from R_i and determining how many inner periods t_j can occur in the remaining time, bounded by the maximum number of such arrivals m_j. (This new definition of I_i is a generalisation of the previous one. For tasks in which $t_j = T_j$, i.e., standard periodic or sporadic tasks, it simplifies to the definition in test 16 (Audsley et al., 1993, p.290).)

A further generalisation of the interference definition in test 17 again allows for release jitter (Audsley et al., 1993, p.291):[4]

$$
\begin{aligned}
(19)\quad \text{for all tasks } i,\quad & R_i \leq D_i\ , \\
\text{where}\quad & R_i = J_i + r_i \\
\text{and}\quad & r_i = C_i + B_i + I_i \\
\text{and}\quad & I_i = \sum_{j=1}^{i-1} \left(\min \left(\left\lceil \frac{J_j + r_i - T_j \left\lfloor \frac{J_j + r_i}{T_j} \right\rfloor}{t_j} \right\rceil , m_j \right) \right. \\
& \qquad \left. + \, m_j \left\lfloor \frac{J_j + r_i}{T_j} \right\rfloor \right) C_j\ .
\end{aligned}
$$

Other extensions to this method allow for more accurate characterisations of implementation overheads due to **tick scheduling** (Burns et al., 1995). Figure 4 shows the special symbols used in this case.

[4] We use a different definition of D_i than Audsley et al. (1993, p.285), starting from *arrival* instead of release time.

S_i	Tick scheduling overheads encountered while an invocation of task i is runnable.
T_{tic}	Polling period of tick scheduler.
C_{int}	Computation time required to service an interrupt (excluding the interrupt routine itself).
C_{ql}	Computation time required to 'awaken' the first suspended task (i.e., move it from the delay queue to the ready queue).
C_{qs}	Computation time required to awaken a subsequent suspended task (assumed to be significantly smaller than handling the first one).

Figure 4: Tick scheduler implementation characteristics (Burns et al., 1995).

Although a tick scheduler can be modelled as a high-priority periodic process, this has been found to be overly pessimistic because the worst case execution of the scheduler, when all tasks need to be moved to the ready queue simultaneously, occurs rarely (Burns et al., 1995, p.477) (Burns and Wellings, 1995, p.715). The following test is more precise. It assumes that periodic tasks are awoken by a polling tick scheduler with period T_{tic}. Thus a periodic task i will experience release jitter proportional to the phase difference between its own period T_i and that of the scheduler (Burns et al., 1995, p.479):

$$J_i = T_{tic} - gcd(T_{tic}, T_i) \ , \quad \text{for periodic task } i.$$

For a sporadic task i initiated by an external interrupt there is no release jitter (although the overheads of implementing the interrupt handler must be included in C_i). However a sporadic task whose arrival is signalled by another task may have to wait for up to T_{tic} time units to be alerted to the event by the scheduler:

$$J_i = T_{tic} \ , \quad \text{for 'polled' sporadic task } i.$$

Another factor to be considered is the delay caused to an invocation of task i by occurrences of the tick scheduler (Burns et al., 1995, p.478) (Tindell et al., 1994, §6). Task i runs for up to r_i time units. The tick scheduler can be invoked

$$L_i = \left\lceil \frac{r_i}{T_{tic}} \right\rceil$$

times during this interval. We can also determine that the scheduler will be required to move some task j from the delay to ready queues up to

$$K_i = \sum_{j=1}^{n} \left\lceil \frac{\left\lceil \frac{r_i}{T_{tic}} \right\rceil T_{tic}}{T_j} \right\rceil$$

times while i is running. This is the interval $\lceil r_i/T_{tic} \rceil T_{tic}$ in which the tick scheduler may be invoked divided by the interarrival time T_j of j. (This assumes each task j suspends itself only at the end of each invocation.) The overall interference caused by the tick scheduler to an invocation of task i is then

$$S_i = L_i C_{int} + \min(K_i, L_i) C_{ql} + \max(K_i - L_i, 0) C_{qs} \ .$$

The first term is the basic cost of handling L_i interrupts. The second term is the overhead of moving 'first' tasks, if any. This is the (long) time C_{ql} to move each one multiplied by the number to be shifted; this may be as low as K_i times, but no more than L_i. The third term is the cost of moving any remaining task invocations; this is the (short) computation time C_{qs} multiplied by the total number to be moved K_i less the number of 'first' tasks L_i, or 0 if no such tasks remain.

The test is then (Burns et al., 1995, p.479),

$$(20) \qquad \begin{aligned} \text{for all tasks } i, \quad & R_i \leq D_i \ , \\ \text{where} \quad & R_i = J_i + r_i \\ \text{and} \quad & r_i = C_i + B_i + I_i + S_i \\ \text{and} \quad & I_i = \sum_{j=1}^{i-1} \left\lceil \frac{\left\lceil \frac{r_i}{T_{tic}} \right\rceil T_{tic}}{T_j} \right\rceil C_j \ . \end{aligned}$$

The response time now includes an explicit allowance S_i for tick scheduling overheads. The interference term replaces the simple bound from test 17 with an exact bound expressed in terms of how many times the tick scheduler can invoke each higher-priority task j during time interval r_i.

Burns and Wellings (1995) present a further variant of this approach in which the overheads of the initial and final context switch associated with each task invocation, and the overheads of implementing the interrupt handler associated with a sporadic task, are expressed explicitly, rather than being incorporated into each C_i. They also discuss the practicalities of dealing with these and other system timing characteristics in an Ada programming environment. In particular, they show how to factor application-specific characteristics into the schedulability test, such as tasks with worst-case computation times that vary according to a known pattern, and task sets that never encounter the worst-case situation in which they all arrive simultaneously.

w_i Level i **busy period**, the longest interval during which tasks of priority equal or greater than i are continuously executing.

Figure 5: Task characteristic defined for tasks with arbitrary deadlines (Tindell and Clark, 1994, §7).

4.4 Arbitrary deadlines

The following tests apply to communicating periodic, sporadic and, in some cases, sporadically-periodic tasks, with arbitrary deadlines that may *exceed* their interarrival times. Special notation used in this situation is shown in Figure 5.

Static-priority scheduling All of the tests below can be used for any static-priority scheduling policy, including rate and deadline-monotonic scheduling. Unique static priorities are assumed (Lehoczky, 1990, p.202).

Lehoczky (1990) showed that a processor-utilisation based schedulability test can be established in terms of a level i **busy period**, i.e., the total time for which tasks of priority level i or higher execute continuously. When deadlines may exceed periods there can be more more than one incomplete invocation of some task i in existence at a time. The computational model adopted assumes that later invocations are delayed until all earlier ones have completed (Tindell et al., 1994, §2).

Firstly, define

$$w_i(x,t) = \sum_{j=1}^{i-1} C_j \left\lceil \frac{t}{T_j} \right\rceil + xC_i$$

to be the duration of the level i busy period for up to x invocations of task i and however many invocations of higher-priority tasks j can occur in a window of duration t time units. (Lower-priority tasks are ignored (Lehoczky, 1990, p.203), so blocking time is not considered by this test.) From this we can determine the processor utilisation by these tasks, in this window, as follows (Lehoczky, 1990, p.203):

$$U_i(x,t) = \min_{u \le t} \frac{w_i(x,u)}{t} \ .$$

The minimum such utilisation value represents the time at which the x^{th} invocation of task i is completed (Lehoczky, 1990, p.203).

The maximum number of invocations of task i that occur in a level i busy period is then (Lehoczky, 1990, p.204)

$$N_i = \min\{x \,|\, U_i(x, xT_i) \leq 1\} \ .$$

This is the smallest number of invocations x such that the level i utilisation, within x periods T_i, is less than 100%. That is, all x invocations of task i have completed within this interval.

The schedulability test is then,

(21) for all tasks i, $\max_{x \leq N_i} U_i(x, (x-1)T_i + D_i) \leq 1$.

For each task i, the test checks that the maximum processor utilisation for all N_i invocations of task i occurring in a level i busy period is less then 100%, when the x^{th} such invocation is required to complete before its deadline. The deadline, measured from the start of the busy period, for the x^{th} invocation of task i is D_i plus the x^{th} arrival time, which is $x-1$ times period T_i.

Tindell et al. (1994) then extended this result to define a response-time test that includes blocking times. Their schedulability test defines a worst-case response time R_i in terms of a busy period window w_i for a particular task invocation:

(22) for all tasks i, $R_i \leq D_i$,

$$\text{where} \quad R_i = \max_{q=0,1,2,\ldots} (w_i(q) - qT_i)$$

$$\text{and} \quad w_i(q) = (q+1)C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{w_i(q)}{T_j} \right\rceil C_j \ .$$

Term $w_i(q)$ represents the duration of the level i busy period that includes the $(q+1)^{\text{th}}$ overlapping invocation of task i, i.e., covering the execution of this task invocation and q preceding (incomplete) invocations. Thus the response time for any invocation of task i is found by taking the window $w_i(q)$ that includes this invocation, and subtracting the part of the window qT_i due to the q previous invocations. The worst-case response time R_i is then the maximum such value for any q. The test is bounded, however, because it can stop when a value of q is reached such that $w_i(q) \leq (q+1)T_i$, i.e., when the q^{th} occurrence of task i completed *before* the next arrival. This marks the end of the busy period for task i (at which time a lower priority task is free to run).

In calculating $w_i(q)$, the first term $(q+1)C_i$ is the execution time required by this invocation and its predecessors. The blocking term B_i assumes that the priority ceiling protocol is used and equals the longest critical section executed by a task of lower priority than i where the shared object has a priority ceiling at least as high as the priority of task i. While task i is 'busy' a lower-priority task can block the *whole* set of incomplete invocations of i only once (Tindell et al., 1994, pp.138-9)! The final term defines the preemption time encountered by task i due to higher-priority tasks j in the interval $w_i(q)$, i.e., the worst-case number of arrivals $\lceil w_i(q)/T_j \rceil$ of task j during $w_i(q)$, multiplied by its worst-case execution time C_j. As for test 16, the recursive equation can be solved iteratively (Tindell et al., 1994, p.138).

Numerous variants of test 22 have been proposed. Release jitter can be accounted for with an extension analogous to that introduced in test 17 (Tindell and Clark, 1994, p.120) (Tindell et al., 1994, §4):

(23) for all tasks i, $R_i \leq D_i$,

$$\text{where} \quad R_i = \max_{q=0,1,2,\ldots} (J_i + w_i(q) - qT_i)$$

$$\text{and} \quad w_i(q) = (q+1)C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{J_j + w_i(q)}{T_j} \right\rceil C_j \ .$$

Sporadically periodic tasks can also be incorporated (Tindell et al., 1994, §5). Here we need to know the number P_i of complete, outer invocations of sporadically periodic task i active at the beginning of the window of interest. For the q^{th} successive, incomplete invocation of i there can be

$$P_i = \left\lfloor \frac{q}{m_i} \right\rfloor$$

such arrivals. Similarly, the number p_i of remaining incomplete inner invocations of i can be determined. For the q^{th} instance of i this must be q less the number of incomplete i tasks due to whole outer arrivals, i.e.,

$$p_i = q - P_i m_i \ .$$

To account for higher-priority, preemptive, sporadically-periodic tasks j, we must allow for the number of arrivals they may have in the window of interest. Allowing for release jitter, let

$$F_j = \left\lceil \frac{J_j + w_i(q)}{T_j} \right\rceil - 1$$

be the number of complete outer periods of higher-priority task j occurring in window $w_i(q)$, less the last one which can be treated as a simple periodic task (Tindell et al., 1994, p.143). Similarly, let

$$f_j = \left\lceil \frac{J_j + w_i(q) - F_j T_j}{t_j} \right\rceil$$

be a bound on the number of inner invocations of j in the remaining part of the window, i.e., the amount of time remaining once the outer invocations are removed, divided by the inner period t_j.

C_i^D The worst case computation time required by task i to complete all externally observable events: $C_i^D \leq C_i^T$ and $C_i^D \leq R_i^D$.

C_i^T The worst case computation time required by task i to complete all activity: $C_i^T \leq R_i^T$ but C_i^T may exceed D_i.

R_i^D The worst case time by which task i must complete all externally observable events: $R_i^D \leq R_i^T$ and $R_i^D \leq D_i$.

R_i^T The worst case time by which task i must complete all activity: R_i^T may exceed D_i.

Figure 6: Task characteristics defined for tasks with deadlines prior to completion times (Burns et al., 1994).

The test then becomes,

$$\begin{aligned}(24)\quad \text{for all tasks } i,\quad & R_i \leq D_i \ , \\ \text{where}\quad & R_i = \max\nolimits_{q=0,1,2,\ldots} (J_i + w_i(q) - p_i t_i - P_i T_i) \\ \text{and}\quad & w_i(q) = (P_i m_i + p_i + 1)C_i + B_i \\ & \qquad + \sum_{j=1}^{i-1} (\min(m_j, f_j) + F_j m_j) C_j \ .\end{aligned}$$

In calculating R_i we subtract all the previous, incomplete outer F_i and inner f_i arrivals of i multiplied by their interarrival times T_i and t_i, respectively. The recursive definition of $w_i(q)$ is (a) the number of outer $P_i m_i$ and inner p_i invocations of i, plus the current invocation, multiplied by its execution time C_i, plus (b) the blocking time B_i, plus (c) the number of preemptions from higher-priority tasks j multiplied by their worst case execution time C_j. The number of preemptions is determined by the number of arrivals of j due to complete outer periods $F_j m_j$ and the least upper bound on the remaining number of inner arrivals $\min(m_j, f_j)$.

Another variant of test 22 allows for the case where each task must complete all externally observable activity before the deadline D_i, but internal activities can continue *after* the deadline has elapsed (Burns et al., 1994, §3). Task behaviour is divided into two parts, that which must be completed by the deadline, and 'local' actions that may be performed after the deadline. Special symbols used in this case are shown in Figure 6. Schedulability test 22 then becomes,

(25) for all tasks i, $R_i^D \leq D_i$,

$$\text{where} \quad R_i^D = \max_{q=0,1,2,\ldots} \left(w_i(q) - qT_i \right)$$

$$\text{and} \quad w_i(q) = qC_i^T + C_i^D + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{w_i(q)}{T_j} \right\rceil C_j^T \ .$$

The definition of window $w_i(q)$ requires the first q occurrences of i to have completed their entire computation time C_i^T, but it is sufficient for the $(q+1)^{\text{th}}$ occurrence to have completed only C_i^D units of execution time before the deadline. (Interference due to higher-priority tasks j still depends on their total execution times C_j^T. Whether j is performing externally observable behaviour or not it still prevents task i from progressing.)

Tindell and Clark (1994) define an extension of tests 22 and 20 to allow for tick scheduling overheads when deadlines exceed periods and the task set contains only sporadic and periodic tasks.

In this case the tick scheduler can be invoked

$$L_i = \left\lceil \frac{w_i(q)}{T_{tic}} \right\rceil$$

times during busy period $w_i(q)$. We can also determine that the scheduler will be required to move a task j from the delay to dispatch queues up to

$$K_i = \sum_{j=1}^{i-1} \left\lceil \frac{J_j + w_i(q)}{T_j} \right\rceil$$

times while overlapping invocations of i are runnable.[5] The overall interference S_i caused by the tick scheduler to an invocation of task i is calculated as for test 20. The test is then (Tindell and Clark, 1994, p.121),

(26) for all tasks i, $R_i \leq D_i$,

$$\text{where} \quad R_i = \max_{q=0,1,2,\ldots} \left(J_i + w_i(q) - qT_i \right)$$

$$\text{and} \quad w_i(q) = (q+1)C_i + \sum_{j=1}^{i-1} \left\lceil \frac{J_j + w_i(q)}{T_j} \right\rceil C_j + S_i \ .$$

(Blocking time has been ignored in this test.)

A further extension to test 24 then allows for sporadically periodic tasks and tick scheduling (Tindell et al., 1994, §6). The values for P_i, p_i, F_j, f_j are

[5]The formula for K_i used here by Tindell and Clark (1994, p.121), sums over higher-priority tasks, rather than *all* tasks. This differs from all other definitions of this form (Tindell et al., 1994, p.144) (Burns and Wellings, 1995, p.717) (Burns et al., 1995, p.478).

calculated as for test 24. The worst case number of times tasks move from the pending to ready queues during $w_i(q)$ is (Tindell et al., 1994, p.144)

$$K_i = \sum_{j=1}^{n} \min(m_j, f_j) + m_j F_j \ ,$$

i.e., the arrivals $m_j F_j$ due to complete outer arrivals of j, plus the least upper bound on the possible inner arrivals in the remaining time $\min(m_j, f_j)$. Values of L_i and S_i are calculated as per test 20. The test is then,

$$\begin{aligned}(27) \quad \text{for all tasks } i, \quad & R_i \le D_i \ , \\ \text{where} \quad & R_i = \max_{q=0,1,2,\ldots} (J_i + w_i(q) - p_i t_i - P_i T_i) \\ \text{and} \quad & w_i(q) = (P_i m_i + p_i + 1) C_i + B_i \\ & \quad + \sum_{j=1}^{i-1} (\min(m_j, f_j) + F_j m_j)\, C_j + S_i \ .\end{aligned}$$

5 Schedulability tests summary

The tables in this section summarise the major capabilities of the schedulability tests presented in this report.

Table 1 shows the types of tasks that the tests can deal with. The column labelled ‘sporadic’ is for truly sporadic tasks with no explicit task server. Presumably *any* of the tests would allow a sufficiently-determined programmer to implement a sporadic task server as a periodic task, but only tests 11 and 12 make *explicit* allowance for the possibility. Park et al. (1996, p.61) briefly mention that test 7 is suitable for use with a sporadic server. Gomaa (1993, §11.4.8), when discussing test 11, refers to sporadic tasks initiated by external interrupts, so we infer that the test is suitable for general sporadic tasks. Lehoczky (1990) and Manabe and Aoyagi (1995), when discussing tests 21 and 6, respectively, refer to periodic tasks only, but we see no impediment in applying these tests to sporadic tasks.

Not shown in table 1 are aperiodic tasks handled by the **priority exchange** or **deferrable server** algorithms (Sprunt et al., 1989) because (ideally!) any soft real-time tasks absorb idle time only and thus do not feature in these schedulability tests. (In fact both algorithms may introduce some degree of interference, and the priority exchange algorithm even makes the task set un**stable**!)

Table 2 shows the scheduling policies assumed by the tests. Tests that allow any static priority allocation have been marked as supporting rate and deadline monotonic scheduling too because such a capability can be used to construct a

	Task type			
Test	periodic	sporadic	sporadic with server	sporadically periodic
1	✓			
2	✓			
3	✓			
4	✓			
5	✓	✓		
6	✓	✓		
7	✓		✓	
8	✓	✓		
9	✓			
10	✓			
11	✓	✓	✓	
12	✓		✓	
13	✓			
14	✓			
15	✓			
16	✓	✓		
17	✓	✓		
18	✓	✓		✓
19	✓	✓		✓
20	✓	✓		
21	✓	✓		
22	✓	✓		
23	✓	✓		
24	✓	✓		✓
25	✓	✓		
26	✓	✓		
27	✓	✓		✓

Table 1: Types of tasks modelled

rate or deadline monotonic priority allocation if desired. None of the tests in our survey assumed **least laxity** scheduling.

Table 3 shows the shared variable locking protocols supported. Those tests with no check marks do not consider task communication at all. In a worst case scenario the behaviour of the **priority ceiling** and **ceiling locking** protocols is identical, so those tests designed for the priority ceiling protocol have been shown as suitable for both.

Burns et al. (1994) do not state which locking protocol is anticipated for test 25 but, given the test's lineage, it is safe to assume suitability for priority ceiling locking. Although Tindell and Clark (1994) do not include a blocking term in test 26, there seems to be no impediment to easily making such an allowance. Similarly, test 21 could be easily extended to consider blocking times. Park et al. (1996, p.62) claim that test 7 can be easily extended with a

	Scheduling policy				
	Static priorities			Dynamic priorities	
Test	rate monotonic	deadline monotonic	any static allocation	earliest deadline first	least laxity
1	✓				
2	✓				
3	✓				
4				✓	
5		✓			
6		✓			
7	✓	✓	✓		
8	✓	✓	✓		
9	✓				
10	✓				
11	✓				
12	✓				
13				✓	
14				✓	
15				✓	
16	✓	✓	✓		
17	✓	✓	✓		
18	✓	✓	✓		
19	✓	✓	✓		
20	✓	✓	✓		
21	✓	✓	✓		
22	✓	✓	✓		
23	✓	✓	✓		
24	✓	✓	✓		
25	✓	✓	✓		
26	✓	✓	✓		
27	✓	✓	✓		

Table 2: Scheduling policies modelled

blocking term for the priority ceiling protocol but, since this is not demonstrated in their paper, we have not marked this in the table.

Table 4 shows the allowed relationships of deadlines to interarrival times. Tests that allow deadlines less than or equal to periods can, of course, always handle the case of deadlines equal to periods, and tests that allow arbitrary deadlines can handle all three situations.

6 Conclusions

Schedulability tests have progressed in sophistication (and complexity!) considerably in the last few years. As the tests have moved from academia to industrial applications, they have been forced to increase their scope. The concept of

	Locking protocol				
Test	priority ceiling	ceiling locking	dynamic priority ceiling	kernelised monitor	stack resource
1					
2					
3					
4					
5					
6					
7					
8					
9	✓	✓			
10	✓	✓			
11	✓	✓			
12	✓	✓			
13				✓	
14			✓		
15					✓
16	✓	✓			
17	✓	✓			
18	✓	✓			
19	✓	✓			
20	✓	✓			
21					
22	✓	✓			
23	✓	✓			
24	✓	✓			
25	✓	✓			
26					
27	✓	✓			

Table 3: Locking protocols modelled

sporadically periodic tasks, and the addition of tick scheduling overheads into some of the later tests, were both in response to the need for more accuracy in particular practical applications. Other application-specific characteristics have been used to customise tests (Burns and Wellings, 1995) and variations have been suggested to remove the simplifying assumption that context-switching overheads are accounted for in task execution times (Sha et al., 1991, §3). Another area of investigation is the addition of offsets to task release times so that the worst-case scenario in which all tasks want to be released simultaneously never occurs. The analysis required for this is surprisingly complex, and an exact schedulability test has proven to be computationally infeasible (Tindell, 1994, p.11)!

Indeed, as the tests have become more and more complex it is natural to ask

	Deadlines vs. interarrival times		
Test	deadlines equal periods	deadlines less than periods	deadlines greater than periods
1	✓		
2	✓		
3	✓		
4	✓		
5	✓	✓	
6	✓	✓	
7	✓	✓	
8	✓	✓	
9	✓		
10	✓		
11	✓		
12	✓		
13	✓		
14	✓		
15	✓	✓	
16	✓	✓	
17	✓	✓	
18	✓	✓	
19	✓	✓	
20	✓	✓	
21	✓	✓	✓
22	✓	✓	✓
23	✓	✓	✓
24	✓	✓	✓
25	✓	✓	✓
26	✓	✓	✓
27	✓	✓	✓

Table 4: Relationship of deadlines to interarrival times

if they can be automated. In fact tools to do so are already appearing (Audsley et al., 1995, p.192). For instance, the STRESS scheduling simulator developed by the University of York (Audsley et al., 1994) has a number of in-built feasibility tests. The SEW toolset, developed at Carnegie Mellon University (Strosnider, 1995), is even more powerful, with an emphasis on checking schedulability of large-scale, multimedia systems.

Furthermore, some authors have directly addressed the computational issues involved in evaluating the tests. Audsley et al. (1991) presented an algorithm for implementing a sufficient and necessary test for deadline-monotonic scheduling. Concerned by the difficulty of assessing schedulability for dynamic scheduling policies, Ripoll et al. (1996, §4) recently proposed a new, efficient testing algorithm that works by simulating earliest deadline first scheduling. The tests developed by Manabe and Aoyagi (1995) were motivated by a desire to reduce computational complexity.

We have considered only preemptive scheduling here, but there is currently a renewed interest in *non*-preemptive scheduling policies (Burns and Wellings, 1996). Although such policies usually have lower processor utilisation than preemptive ones, they are cheaper to implement because mutually-exclusive access to shared variables is guaranteed automatically, without the need for a locking protocol. Also their conceptually simpler run-time behaviour makes this approach attractive in safety-critical applications (Bate et al., 1996).

Consideration of *soft* real-time tasks is another active area of research, aiming to allow non-critical tasks to profitably absorb any processor time left unused by those tasks with hard timing constraints (Davis, 1994).

An especially dramatic development in recent years has been the impact scheduling theory has had on programming language design. Numerous features in the new Ada 95 standard (ISO, 1994) have been included specifically to make it possible to write Ada programs amenable to real-time schedulability testing (Stoyenko and Baker, 1994).

Finally, we note that although we have limited this survey to uniprocessor systems, schedulability techniques for multiprocessor and distributed systems are beginning to emerge (Sha and Sathaye, 1995) (Tindell and Clark, 1994) (Audsley et al., 1993, §4.1) (Joseph and Pandya, 1986, §8) (Shin and Ramanathan, 1994, p.9). Unfortunately multiprocessor scheduling is not simply a matter of extending the uniprocessor methods. For instance, optimal scheduling policies for uniprocessor systems are not necessarily optimal for multiprocessor systems (Audsley et al., 1995, §8.2) and deciding schedulability for multiprocessor systems can be NP-hard (Leung and Whitehead, 1982).

Acknowledgements

I wish to thank the anonymous referees for their many helpful suggestions and pointers, Andy Wellings for advice on the ceiling protocols, Michael Pilling and Jay Strosnider for comments on schedulability testing, and Peter Kearney for reviewing a draft of this report. This work was funded by the Information Technology Division of the Defence Science and Technology Organisation.

References

Audsley, N. and Burns, A. (1990). Real-time system scheduling. Technical Report YCS 134, Department of Computer Science, University of York.

Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A. (1993). Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292.

Audsley, N. C., Burns, A., Davis, R. I., Tindell, K. W., and Wellings, A. J. (1995). Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8:173–198.

Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J. (1991). Hard real-time scheduling: The deadline monotonic approach. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137.

Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J. (1994). Stress: A simulator for hard real-time systems. *Software—Practice & Experience.*

Baker, T. P. (1990). A stack-based resource allocation policy for realtime processes. In *Proc. Real-Time Systems Symposium*, pages 191–200. IEEE Computer Society Press.

Baker, T. P. (1991). Stack-based scheduling of real-time processes. *Real Time Systems*, 3(1):67–99.

Barnes, J. G. P. (1993). *Programming in Ada Plus an Overview of Ada 9X.* Addison-Wesley.

Bate, I. J., Burns, A., and Audsley, N. C. (1996). Putting fixed priority scheduling theory into engineering practice for safety critical applications. In *Proc. Second Real-Time Applications Symposium*, pages 2–10, Boston.

Burns, A., Tindell, K., and Wellings, A. J. (1994). Fixed priority scheduling with deadlines prior to completion. In *Proc. Sixth Euromicro Workshop on Real-Time Systems*, pages 138–142.

Burns, A., Tindell, K., and Wellings, A. J. (1995). Effective analysis for engineering real-time fixed priority schedulers. *IEEE Trans. on Software Engineering*, 21(5):475–480.

Burns, A. and Wellings, A. J. (1990). *Real-Time Systems and their Programming Languages.* Addison-Wesley.

Burns, A. and Wellings, A. J. (1991). Priority inheritance and message passing communication: A formal treatment. *The Journal of Real-Time Systems*, 3:19–44.

Burns, A. and Wellings, A. J. (1995). Engineering a hard real-time system: From theory to practice. *Software–Practice & Experience*, 25(7):705–726.

Burns, A. and Wellings, A. J. (1996). Simple Ada 95 tasking models for high integrity applications. Department of Computer Science, University of York.

Chapman, R., Burns, A., and Wellings, A. J. (1994). Integrated program proof and worst-case timing analysis of SPARK Ada. In *ACM Workshop on language, compiler and tool support for real-time systems.* ACM Press.

Chen, M.-I. and Lin, K.-J. (1990). Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2(4):325–346.

Davis, R. (1994). Dual priority scheduling: A means of providing flexibility in hard real-time systems. Technical Report YCS 230, Department of Computer Science, University of York.

Giering III, E. and Baker, T. (1994). A tool for deterministic scheduling of real-time programs implemented as periodic ada tasks. *ACM Ada Letters*, XIV:54–73.

Gomaa, H. (1993). *Software Design Methods for Concurrent and Real-Time Systems.* Addison-Wesley.

ISO (1994). *Ada Reference Manual: Language and Standard Libraries*, 6.0 edition. International Standard ISO/IEC 8652:1995.

Jones, M. B., Barrera III, J. S., Forin, A., Leach, P. J., Rosu, D., and Rosu, M.-C. (1996). An overview of the Rialto real-time architecture. In *Proc. Seventh ACM SIGOPS European Workshop.*

Joseph, M. and Pandya, P. (1986). Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395.

Kenny, K. and Lin, K.-J. (1991). Measuring and analyzing real-time performance. *IEEE Software*, 8(5):41–49.

Klein, M. H. and Ralya, T. (1990). An analysis of input/output paradigms for real-time systems. Technical Report CMU/SEI-90-TR-19, Software Engineering Institute, Carnegie Mellon University.

Lehoczky, J. P. (1990). Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. Real-Time Systems Symposium*, pages 201–209. IEEE Computer Society Press.

Lehoczky, J. P., Sha, L., and Ding, Y. (1989). The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press.

Leung, J. Y.-T. and Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250.

Lim, S.-S., Bae, Y. H., Jang, G. T., Rhee, B.-D., Min, S. L., Park, C. Y., Shin, H., Park, K., Moon, S.-M., and Kim, C. S. (1995). An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61.

Locke, C. D. (1992). Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *The Journal of Real-Time Systems*, 4:37–53.

Manabe, Y. and Aoyagi, S. (1995). A feasibility decision algorithm for rate monotonic scheduling of periodic real-time tasks. In *Proc. Real-time Technology and Applications Symposium (RTAS'95)*, pages 212–218.

Mercer, C. W. (1992). An introduction to real-time operating systems: Scheduling theory. School of Computer Science, Carnegie Mellon University.

Park, D.-W., Natarajan, S., and Kanevsky, A. (1996). Fixed-priority scheduling of real-time systems using utilization bounds. *Journal of Systems and Software*, 33(1):57–63.

Puschner, P. and Koza, C. (1989). Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176.

Ripoll, I., Crespo, A., and Mok, A. K. (1996). Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19–39.

Sha, L. and Goodenough, J. B. (1990). Real-time scheduling theory and Ada. *IEEE Computer*, 23(4):53–62.

Sha, L., Klein, M. H., and Goodenough, J. B. (1991). Rate monotonic analysis for real-time systems. Technical Report CMU/SEI-91-TR-6, Software Engineering Institute, Carnegie Mellon University.

Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185.

Sha, L., Rajkumar, R., Lehoczky, J., and Ramamritham, K. (1989). Mode change protocols for priority-driven preemptive scheduling. *Journal of Real-Time Systems*, 1(3):243–264.

Sha, L., Rajkumar, R., and Lehoczky, J. P. (1987). Priority inheritance protocols: An approach to real-time synchronisation. Technical Report CMU-CS-87-181, Department of Computer Science, Carnegie Melon University.

Sha, L. and Sathaye, S. S. (1995). Distributed system design using generalized rate monotonic theory. Technical Report CMU/SEI-95-TR-011, Software Engineering Institute, Carnegie Mellon University.

Shaw, A. C. (1989). Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889.

Shin, K. G. and Ramanathan, P. (1994). Real-time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24.

Sprunt, B., Sha, L., and Lehoczky, J. (1989). Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1(1):27–60.

Stoyenko, A. D. and Baker, T. P. (1994). Real-time schedulability-analyzable mechanisms in Ada 9X. *Proceedings of the IEEE*, 82(1):95–107.

Strosnider, J. K. (1995). Performance engineering real-time/multimedia systems. Seminar presentation, Uni. Qld. Contact: Department of Electrical and Computer Engineering, Carnegie Mellon University, `Strosnider@gauss.ece.cmu.edu`.

Taft, S. T. (1992). Ada 9X: A technical summary. *Communications of the ACM*, 35(11):77–82.

Tindell, K. (1994). Adding time-offsets to schedulability analysis. Technical Report YCS 221, Department of Computer Science, University of York.

Tindell, K., Burns, A., and Wellings, A. J. (1994). An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6:133–151.

Tindell, K. and Clark, J. (1994). Holistic schedulability analysis for distributed hard real-time systems. *Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134.

A Glossary of preemptive scheduling terminology

The following terminology has been used when discussing scheduling principles in this article. **Bold** text denotes a cross reference.

Active (or effective) priority (ISO, 1994, §D.1) (Audsley et al., 1993, §2) In static-priority scheduling, a variable **priority** value associated with each task, usually at least as great as the **base priority** of the task. **Priority inheritance** may temporarily raise the active priority of a task above its base priority.

Aperiodic task (Sprunt et al., 1989, p.28) (Audsley and Burns, 1990, §2.1) Some job that **arrives** randomly, typically in response to an external triggering event. Many such arrivals may occur in rapid succession. Hard real-time deadlines are impossible to enforce for aperiodic tasks: they may have soft 'performance' goals only. Also see **sporadic task**.

Arrival (Audsley et al., 1993, §2) The time at which a task invocation wishes to start running. This is the start of each period for a **periodic task** and the occurrence of the triggering event for an **aperiodic** or **sporadic task**.

Base priority (Audsley et al., 1993, §2) In static-priority scheduling, a fixed **priority** permanently associated with each task.

Blocking (Sha et al., 1990, p.1177) (Audsley and Burns, 1990, p.6) A task is blocked if it is ready to run but cannot because some shared variable it requires is locked by a task of *lower* priority (c.f. **preemption**).

(Worst case) blocking time (Audsley et al., 1993, p.285) An upper bound on the duration for which a task may be prevented from executing by a low priority task that has locked a shared resource needed by the higher-priority task. The actual worst-case blocking time is a property of the particular scheduling policy and locking protocol used, as this determines how many times each task invocation may be blocked. Typically, however, the value will be some multiple of the worst-case computation time spent by lower-priority tasks executing code in a critical region (Sha et al., 1990, p.1177), or protected object (ISO, 1994, §17.8.1). Actual computation times for these activities can be determined experimentally (Kenny and Lin, 1991) or, more reliably, through formal analysis of high-level language (Chapman et al., 1994) (Shaw, 1989) (Puschner and Koza, 1989), or assembler (Lim et al., 1995), code.

Busy period (Lehoczky, 1990, p.202) (Tindell et al., 1994, p.137) When task **deadlines** may exceed their inter**arrival** times, the longest contiguous interval of time during which there are one or more unsatisfied **invocation requests** for a particular task.

(Priority) ceiling locking (or emulation) protocol (Stoyenko and Baker, 1994, p.104) (ISO, 1994, §D.1, D.3) A special case of the **stack resource protocol** (Stoyenko and Baker, 1994, p.104) for static-priority scheduling. It modifies **priority inheritance** by (a) statically allocating each shared variable a *ceiling value* defined as the maximum **base priority** of all tasks that may use it, and (b) making each task that locks a shared variable inherit the ceiling value from that variable as its **active priority**. This effectively prevents any task invocation from starting to run until all the shared variables it may want to use are free (ISO, 1994, §D.2.1). Ceiling locking may cause tasks to be blocked in situations where they are not blocked by the **priority ceiling protocol**. However, in the worst case the same degree of blocking is experienced by both protocols and, since schedulability tests assume worst-case scenarios, any schedulability test applicable to the priority ceiling protocol also applies to the ceiling locking protocol. Ceiling locking is simpler to implement than the priority ceiling protocol.

(Worst case) computation time (Audsley and Burns, 1990, §5.1) An upper bound on the amount of processor time required by a task for any invocation. Schedulability tests usually assume that scheduling overheads are included in the computation time for each task (Gomaa, 1993, p.124): each invocation includes one context switch to the task itself, and one switch back to the task it preempted (Burns and Wellings, 1995, p.714) (Burns et al., 1995, p.477); a sporadic task initiated by an external interrupt must incorporate the interrupt handling overhead (Burns et al., 1995, p.479); regular interrupts due to a **tick scheduler** must also be considered (Burns et al., 1995). Actual computation times can be determined experimentally (Kenny and Lin, 1991) or, more reliably, through formal analysis of high-level language (Chapman et al., 1994) (Shaw, 1989) (Puschner and Koza, 1989), or assembler (Lim et al., 1995), code.

Deadline (Tindell et al., 1994, p.149) A fixed time by which a task invocation must have completed its computation, measured relative to its **arrival** time. (Some authorities measure the deadline from **release time** (Audsley and Burns, 1990, §2.1) (Audsley et al., 1993, p.255).) If no specific deadline is provided for a **periodic task** it is assumed to equal the period.

Deadline monotonic (or inverse-deadline) scheduling
(Leung and Whitehead, 1982, p.240) (Lehoczky, 1990, p.202) (Audsley et al., 1991) A static-priority preemptive scheduling policy in which tasks with shorter (relative) **deadlines** have higher priority.

Deferrable server algorithm (Sprunt et al., 1989, §2.2) A method of servicing **aperiodic tasks** when using **rate monotonic scheduling**. It (a) implements a high-priority periodic task to handle aperiodic activities, (b) allows this task to **suspend** itself if it is activated when there are no outstanding aperiodic requests, while preserving its allotted computation time until such a request arrives, and (c) replenishes the computation time for the server task at the start of each period. The algorithm is simple and **stable** for the periodic tasks (Audsley and Burns, 1990, p.12) but allows lower utilisation than the **priority exchange algorithm** (Sprunt et al., 1989, p.35).

Delay (or pending) queue (Burns et al., 1995, p.476) A notional queue in which tasks that have been **suspended** wait, usually ordered by the time at which they become ready again. A task suspends itself by executing a 'delay' statement. This removes it from the **ready queue** and places it in the delay queue. It is removed from the delay queue and placed back in the ready queue by the scheduler, typically following a clock-driven interrupt. Also see **tick scheduling**.

Earliest deadline first (or deadline driven) scheduling (Liu and Layland, 1973, §7) (Audsley and Burns, 1990, §3.2.1) A dynamic-priority preemptive scheduling policy in which the task with the earliest (absolute) deadline from the current moment has the highest priority.

Feasible task set (Audsley and Burns, 1990, §2.2) A **task set** for which a schedule exists.

Interference (Audsley et al., 1993, §3) The degree of time for which a task invocation may be **preempted**.

Invocation (or job) request (Audsley et al., 1993, §2) A nominal event marking the **arrival** of a task. This 'event' involves the task requesting computational resources from the scheduler.

Jitter (Giering III and Baker, 1994, p.55) (Locke, 1992, p.49) Variability in the actual separation between invocations of a **periodic task** from its intended period due to its ability to be scheduled anywhere between its **arrival** time and **deadline**. Jitter can be reduced by shortening the task deadline, but this may make the task set in**feasible** (Giering III and Baker, 1994, p.55). Also see **release jitter**.

Kernelised monitor protocol (Chen and Lin, 1990, §2.2) A shared variable access method for use with **earliest deadline first** scheduling. It assumes all tasks in critical sections are not preemptible. It can lead to low **processor utilisation** if there are critical sections with long execution times.

Least laxity scheduling (Audsley and Burns, 1990, §3.2.2) (Jones et al., 1996, §5) A dynamic-priority preemptive scheduling policy in which high priority is given to task invocations with the smallest difference between their upcoming absolute deadline and the remaining computation time they require.

Necessary test (Audsley et al., 1993, p.284) A schedulability test that fails only in**feasible task sets** (c.f. **sufficient test**).

Optimal scheduling policy (Audsley and Burns, 1990, p.6) (Manabe and Aoyagi, 1995, p.213) A scheduling policy which can produce a schedule for any **feasible task set**. Among static-priority policies **rate monotonic scheduling** is optimal for independent periodic tasks with deadlines equal to their periods (Liu and Layland, 1973, p.178) (Giering III and Baker, 1994, p.55) and **deadline monotonic scheduling** is optimal for periodic tasks with deadlines less than their period (Leung and Whitehead, 1982, §2) (Audsley et al., 1993, p.284) (Tindell and Clark, 1994, p.134) (Burns and Wellings, 1995, p.708). Among dynamic-priority policies **earliest deadline first** is optimal for independent periodic tasks with deadlines equal periods (Liu and Layland, 1973, p.186) (Giering III and Baker, 1994, p.55).

Period transformation (Sha and Goodenough, 1990, p.56) (Audsley and Burns, 1990, §3.3.1) Since the period of a task is not necessarily a measure of its relative importance, period transformation techniques seek to make a **task set** suitable for **rate monotonic scheduling** by breaking highly important tasks with long worst-case **computation times** into smaller, more frequently scheduled pieces.

Periodic task (Audsley and Burns, 1990, §2.1) Some job that **arrives** at fixed intervals (c.f. **aperiodic task**). A periodic task is usually characterised by its period, **computation time**, **deadline** and shared variable access (Audsley et al., 1993, §2) requirements.

Preemption (Audsley et al., 1993, §2) A task is preempted if it is ready to run but cannot because a task of *higher* priority is running (c.f. **blocking**).

Priority (Burns and Wellings, 1991, p.19) An allocation of (usually unique) numbers to tasks, used by a scheduler to determine which ready task should run. Tasks with higher priorities run in preference to those with lower priorities.

Priority ceiling protocol (Sha and Goodenough, 1990, p.57) (Sha et al., 1990, §IV) (Audsley and Burns, 1990, §4.4) A method of preventing **priority inversion** and bounding **blocking** times, suitable for use with static-priority scheduling policies. It extends **priority inheritance** by (a) statically allocating each shared variable a *ceiling* value defined as the maximum **base priority** of all tasks that may use it, and (b) allowing a task to lock a variable only if its **active priority** is higher than the ceiling value of *any* variable currently locked by any other task (Sha et al., 1990, p.1178). In the worst case with this protocol each task invocation is blocked at most once (c.f. **priority inheritance protocol**), although it can be 'blocked' even if it does not access any shared variables (Sha and Goodenough, 1990, p.58)! A high-priority task may start running even when some of the resources it will need are locked by lower-priority tasks (Sha and Goodenough, 1990, pp.56–7) (c.f. **ceiling locking protocol**). The protocol also has the beneficial side-effect of preventing deadlocks. A dynamic version of the priority ceiling protocol is available for use with **earliest deadline first** scheduling, where the ceiling value for each variable changes according to upcoming deadlines (Chen and Lin, 1990, pp.332–3) (c.f. **stack resource protocol**). Also see **ceiling locking protocol**.

Priority exchange algorithm (Sprunt et al., 1989, §2.2) A method of servicing **aperiodic tasks** when using **rate monotonic scheduling**. It (a) uses a periodic server task to handle aperiodic requests, (b) allows the server to **suspend** itself if there are no outstanding aperiodic requests by swapping its priority with the highest-priority periodic task, and (c) replenishes the available computation time (and priority) of the server at the start of its period. The algorithm has greater responsiveness than a simple periodic polling server but makes the periodic task set un**stable** (Audsley and Burns, 1990, p.11) (c.f. **deferrable server algorithm**).

Priority inheritance protocol (Sha et al., 1990, §III) (Chen and Lin, 1990, p.328) A method of preventing **priority inversion**. This is done by temporarily raising the **active priority** of a task locking a shared variable to the maximum **base priority** of any task **blocked** on that variable. In the worst case with this protocol a task is blocked at most once for *each* shared variable it attempts to access (c.f. **priority ceiling protocol**).

Priority inversion (Sha and Goodenough, 1990, p.57) (Sha et al., 1990, §II) (Audsley and Burns, 1990, §4.1) The situation where (a) a task of low priority is **blocking** one of higher priority and (b) a task of medium priority then **preempts** the low-priority one, thus further blocking the high priority task and 'inverting' the desired relationship about which task runs next. Priority inversion can be prevented by guaranteeing that tasks never attempt to access the same variable simultaneously (Audsley and Burns, 1990, §4.2) or by using a **priority inheritance** protocol. More generally, 'priority inversion' can also refer to situations where the scheduler is undertaking actions on behalf of low-priority tasks while preempting a high-priority task (Burns et al., 1995, §II).

Processor utilisation (Chen and Lin, 1990, p.327) (Manabe and Aoyagi, 1995, p.213) The overall percentage of time that a task (or **task set**) may require access to the processor. It is defined as the ratio of **computation time** divided by inter**arrival** time.

Rate monotonic scheduling (Liu and Layland, 1973, p.178) (Sha et al., 1991) A static-priority preemptive scheduling policy in which tasks with shorter periods have higher priorities. Also see **period transformation**.

Ready (or run or dispatch) queue (Burns et al., 1995, p.476) (Audsley et al., 1993, §2) (ISO, 1994, §D.2.1) A notional queue of task invocations that are ready to run given sufficient resources, usually ordered by their priorities.

Release jitter (Audsley et al., 1993, §4) (Tindell et al., 1994, p.134) The difference between the **arrival** time of a task invocation and its **release time**. This gap may be due to the scheduler taking some time to recognise task arrival (Audsley et al., 1993, p.287) or due to a task awaiting input from the environment before it can start (Audsley et al., 1993, §4.1).

Release time (Audsley et al., 1993, §2) The time at which the scheduler acknowledges that a task invocation that has **arrived** is ready to run, typically by placing it in the **ready queue**. (The task does not necessarily begin running at this time, however.)

(Worst case) response time (Tindell et al., 1994, p.150) The time at which a task invocation completes all activity, measured relative to its **arrival time**. For a task to be schedulable its worst case response time must not exceed its **deadline**. (A relaxation of this constraint is to require only that the task completes all *observable* activity by the deadline (Burns et al., 1994).)

Shared resources (ISO, 1994, §D.2.1) (Audsley and Burns, 1990, p.6) Any resource needed by more than one task. Typically this includes all shared variables and the processor. The processor is a preemptible resource because, once allocated to a task, it may temporarily be allocated to another. Access to shared variables is made non-preemptible in uniprocessor scheduling by protecting them with semaphores or monitors: once allocated to a task a variable cannot be allocated to another until unlocked (although the task accessing the variable may itself be **preempted** by a higher-priority task that does not use the variable).

Sporadic server algorithm (Sprunt et al., 1989, §3.1) (Sha and Goodenough, 1990, pp.56,60) A method of handling **sporadic tasks** using a preemptive server. For **rate monotonic scheduling** with sporadic task **deadlines** at least as great as their interarrival time it is the same as the **deferrable server algorithm** except that the available computation time is replenished after the interarrival time has elapsed from the last time a request was serviced, rather than periodically (Sprunt et al., 1989, p.37). When sporadic task deadlines are less than their interarrival time the algorithm allows the **base priorities** for sporadic servers to be greater than their rate monotonic value (Sprunt et al., 1989, §4).

Sporadic task (Sprunt et al., 1989, p.28) (Audsley and Burns, 1990, §2.1) A special case of **aperiodic tasks** where **arrivals** have a known minimum separation. This makes it possible to enforce hard deadlines. A sporadic task is characterised by this minimum separation and its **computation time**, **deadline** and shared variable access (Audsley et al., 1993, §2) requirements. For the purposes of worst case analysis sporadic tasks are usually treated like **periodic tasks** with a period equal to the minimum separation (Gomaa, 1993, §11.4.6).

Sporadically periodic task (Tindell et al., 1994, p.134) (Audsley et al., 1993, §5) A combination of both **periodic** and **sporadic task** behaviour in which the task initially arrives sporadically but then rearrives periodically a fixed number of times. A sporadically periodic task is characterised by its **computation time**, 'outer' minimum arrival separation, 'inner' period, **deadline** and shared variable access requirements.

Stability (Sha and Goodenough, 1990, p.56) (Gomaa, 1993, p.124) A scheduling policy is stable if, under conditions of **transient overload**, it is the lowest priority tasks that miss their deadlines. **Rate monotonic** scheduling is stable (Gomaa, 1993, p.124) whereas **earliest deadline first** is not (Audsley and Burns, 1990, p.11).

Stack resource protocol (Baker, 1990, §3.2) (Baker, 1991) A shared variable access protocol, suitable for both dynamic and static scheduling policies. It (a) associates a static *preemption level* with each task, (b) maintains a dynamic *current ceiling* for each shared variable, (c) whenever a variable is accessed, raises the ceiling of the variable to be at least as great as the preemption level of *any* task that may use the variable, and (d) allows a task invocation to start running only when its preemption level exceeds the current ceiling of *every* shared variable. This guarantees that tasks begin only when all the resources they may need are available.

Sufficient test (Audsley et al., 1993, p.284) A schedulability test that passes only **feasible task sets** (c.f. **necessary test**). It may *fail* some schedulable task sets, however.

Suspension (Audsley and Burns, 1990, p.6) A suspended task is one that has made itself temporarily inactive by executing a 'delay' statement. Suspended tasks reside in the **delay queue**.

Task set (Audsley et al., 1993, p.284) A set of scheduling requirements typically consisting of several **periodic** and/or **sporadic task** requirements.

Tick (or timer-driven) scheduling (Burns et al., 1995, §II) (Tindell et al., 1994, §6) A polling method of implementing 'delay' statements. A tick scheduler executes periodically, controlled by a regular clock interrupt. When activated it moves tasks from the **delay queue** to the **ready queue** if their delay interval has passed.

Transient overload (Audsley and Burns, 1990, §3.3) (Sha and Goodenough, 1990, p.56) The situation where one or more task invocations miss their deadlines. This may be due either to the programmer underestimating the worst-case **computation time** requirement for a task or **sporadic task arrivals** occurring at greater than their specified frequency.