Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling

A. Wellings
M. Richardson
A. Burns
N. Audsley
K. Tindell

Real Time Systems Research Group
Department of Computer Science
University of York
England
e-mail: ken@minster.york.ac.uk

ABSTRACT

The paper presents exact schedulability analysis for real-time systems scheduled at run-time with a static priority pre-emptive dispatcher. The tasks to be scheduled are allowed to experience internal blocking (from other tasks with which they share resources) and (with certain restrictions) external blocking — such as waiting for a message to arrive. The analysis presented is more general than that previously published, and subsumed, for example, techniques based on the Rate Monotonic approach. In addition to presenting the theory an existing avionics case study is described and analysed. The predictions that follow from this analysis are seen to in close agreement with the behaviour exhibited during simulation studies.

1. INTRODUCTION

One proposed method of building a hard real time system is from a number of periodic and sporadic tasks, and a common way of scheduling such tasks is by using a static priority pre-emptive scheduler — at run-time the highest priority runnable task is run, pre-empting other lower priority tasks. This scheme was proposed for the Rate Monotonic approach by Liu and Layland [11] where the static priorities for a task is obtained from the period of that task: a short period results in a high priority. Liu and Layland give *a priori* schedulability analysis which determines if a given task set will always meet all deadlines under all possible release timings. The schedulability test given is *sufficient* (*i.e.* all task sets passing the test are guaranteed to be schedulable), but *not necessary* (*i.e.* a task set failing to pass the test is not necessarily unschedulable). Sha, Ding, and Lehoczky give an exact Rate Monotonic test that is both sufficient and necessary [7].

The Rate Monotonic approach has a number of restrictions:

- Tasks are assumed not to interact (*e.g.* tasks do not share resources or communicate)
- The deadline requirement of a task (*i.e.* the time by which the task must complete after being released) is assumed to be equal to the period of that task.
- Tasks are assumed to be periodic — they cannot arrive sporadically.
- Tasks cannot voluntarily suspend themselves

The restriction that tasks cannot interact has been removed by the Priority Ceiling protocol

[16] (and other similar protocols such as the Stack Resource Protocol [4]). A method of allowing sporadic tasks using periodic *servers* is proposed by Lehoczky *et al* [8] (analysis is provided which can guarantee the worst-case response time of a sporadic task). Rajkumar [15] uses external blocking with the Rate Monotonic approach to model the operation of a multi-processor Priority Ceiling protocol, and provide *a priori* schedulability analysis to bound the effects of external blocking [15].

The restriction that the deadline of a task must be equal to the period has not been relaxed for the Rate Monotonic approach. This is perhaps the most important restriction to lift: requiring the deadline of a task to be less than or equal to the period of that task is essential if *jitter* requirements are to be met (*i.e.* the result of a piece of computation must be produced at precise intervals); furthermore, when building distributed systems the deadline of a task often needs to be shortened to allow time for communication between tasks on different processors.

Work by Leung and Whitehead [10] shows that the Deadline Monotonic priority assignment is an optimal one — a task with a short deadline (measured relative to the release time of the task) should have a high priority. A task with a long period but short deadline would have a low priority according to the Rate Monotonic priority assignment, but a high priority according to the Deadline Monotonic priority assignment. Consequently, the Rate Monotonic assignment will be sub-optimal for some task sets. However, before the Deadline Monotonic approach can be used *a priori* analysis must be developed which will allow deadlines to be guaranteed. Such analysis is provided by Joseph and Pandya [5], Lehoczky [9], and Audsley *et al* [1]. All provide sufficient and necessary schedulability tests, differing only in complexity of computation. The basic approach is expanded by Audsley *et al* to permit sporadic task deadlines to be guaranteed without use of the servers required by the Rate Monotonic approach [8].

At this point it is useful to point out that in many cases we wish to pose a different schedulability question to that assumed so far. Instead of asking "in the worst-case will this task meet its deadline?" it is often more useful to ask "what is the longest time this task can ever take to complete?". In particular, the latter question becomes the only one that is meaningful when no deadline requirement can be found — very often when building a distributed system the worst-case time a task takes (*i.e.* the *response time* of the task) forms part of a larger system-wide response time.

This paper is concerned with providing schedulability analysis to give the worst-case response times for a set of periodic and sporadic tasks under any given priority assignment, and scheduled by a static priority pre-emptive scheduler. Section 2 describes the computation model assumed by the analysis. Section 3 provides analysis for guaranteeing the worst-case response times of periodic and sporadic tasks. This section also includes a glossary of notation. Section 4 modifies the analysis to take account of the so-called ‘external blocking’ problem. Section 5 discusses the analysis and applies it to a small avionics case-study.

2. COMPUTATIONAL MODEL

In this paper, unless explicitly mentioned, we consider the scheduling of tasks on a single processor.

A *task i* is assumed to consist of an infinite number of *invocation requests*, each separated by a minimum time T_i. Each invocation is a request to perform a bounded amount of computation C_i, and to lock and unlock semaphores from a bounded set $s(i)$ according to the Priority Ceiling Protocol [16]. Some tasks will have a deadline requirement such that all computation for an invocation must take place before a certain

time measured relative to the invocation request. Deadlines, where required, are assumed to be constant and known *a priori*. The deadline of a task i is denoted $\mathcal{D}_i$.

The arrival of the task will be recognised by the scheduler and the task will be placed in a notional queue of runnable tasks. The task is said to be *released*. A task has a static base priority assigned to it *a priori* (under the operation of the Priority Ceiling Protocol the task may inherit a higher priority). The scheduler chooses to run the highest actual priority runnable task, pre-empting lower priority tasks when necessary. For the moment tasks are assumed to undergo *no* external blocking (*i.e.* cannot suspend themselves mid-execution awaiting an external event).

3. FINDING WORST-CASE RESPONSE TIMES

Before proceeding further we introduce some simple notation in a glossary below. All internal blocking is assumed to be the result of semaphore use. (other synchronisation primitives could be used [2]).

C_i The worst-case computation time required by task i on each release. At run-time we assume that any computation time from 0 to C_i could be required for a single invocation of i

T_i The lower bound on the time between successive arrivals of i. If i is a periodic task then this lower bound will also be the upper bound (*i.e.* the period is fixed and equal to T_i)

$\mathcal{D}_i$ The deadline *requirement* of task i, measured relative to a given release of i. Note that we require $\mathcal{D}_i \leq T_i$.

B_i The worst-case blocking time task i can experience due to the operation of the priority ceiling protocol (or equivalent concurrency control protocol). B_i is normally equal to the longest critical section of lower priority tasks accessing semaphores with ceilings higher than the priority of i.

X_i The worst-case time task i can spend externally blocked in any single invocation.

I_i The worst-case interference a task i can experience. Interference on i is defined as the time higher priority tasks can pre-empt i

D_i The worst-case response time for a task i. For a schedulable task $D_i \leq \mathcal{D}_i$ (if there was no deadline requirement for i we would require that $D_i \leq T_i$).

$hp(i)$ The set of tasks of higher priority than i and that could pre-empt i. Normally this set would include all higher priority tasks — however, for some applications (such as the avionics example considered later), there may be some tasks which could never interfere with a task i; in this case the set $hp(i)$ would not include those tasks.

$lp(i)$ The set of tasks of lower priority than i and that could cause i to block when attempting to lock a semaphore. Normally this set would include all lower priority tasks.

$s(i)$ The set of all semaphores which could be locked by i

$p(i)$ The priority of task i

$c(s)$ The priority ceiling of semaphore s, equal to the priority of the highest

priority task accessing s

$cs(s,i)$ The longest time that task i could take while holding semaphore s

We now turn to the problem of computing the worst-case response time for a task i, denoted D_i. This time can be viewed as a computational 'window': the release of i marks the start of the window, and the completion of i marks the end of the window. The maximum width of the 'window' is D_i. In this 'window' of duration D_i, task i must (at worst) complete an amount of computation equal to C_i, and be delayed when locking semaphores by at most B_i. Additionally, task i could be pre-empted by at most I_i. Therefore we can say that:

$$D_i = C_i + B_i + I_i \tag{1}$$

If a task i has a deadline then we must have $D_i \leq \mathcal{D}_i$.

The worst-case computation time C_i is constant and known *a priori* by some means [13, 14]. The worst-case blocking time B_i is found according to the analysis given by Sha *et al* [16], and is equal to the longest critical section of any lower priority task accessing a semaphore with ceiling of equal or higher priority than task i.

The rest of this section presents analysis to find I_i. Note that similar analysis, cast in a different form, was first produced by Joseph and Pandya [5], and later by Audsley *et al* [1]. The method described below has the advantage that it is easily extended to cover situations such as external blocking.

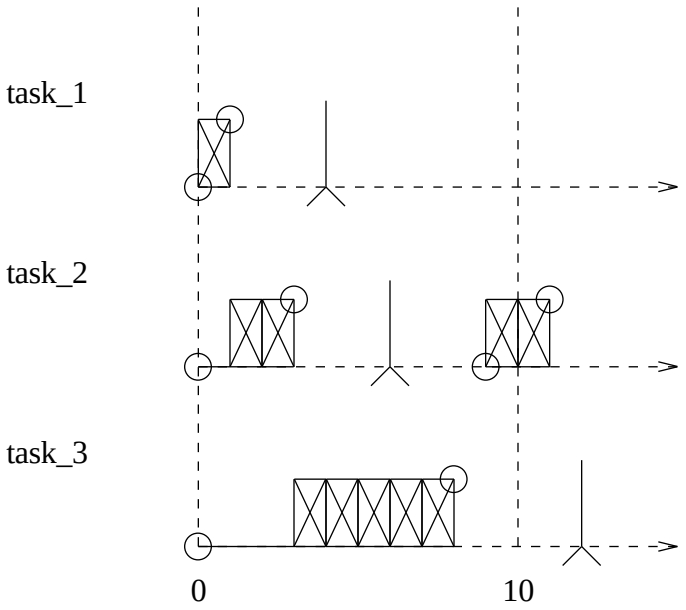


Figure 1

To find I_i first consider Figure 1. The diagram was produced by a tool called *STRESS* [3] written by the Real Time Systems Research Group at York; Appendix 1 describes the notation used in these diagrams. The diagram shows part of a schedule of a system consisting of three tasks, displayed in priority order. Task 1 is a task with worst-case computational requirement of $C_1 = 1$, a deadline of $\mathcal{D}_1 = 4$, and a worst-case inter-arrival time of $T_1 = 50$. Task 2 has $C_2 = 2$, $T_2 = 9$, and $\mathcal{D}_2 = 6$. Task 3 has $C_3 = 5$, $T_3 = 20$, and $\mathcal{D}_3 = 12$. Figure 1 shows the worst-case scheduling point described by Liu and Layland,

where all tasks are released simultaneously (at time 0).

As can be seen from the diagram, task 3 is pre-empted by task 1 for 1 tick and task 2 for 2 ticks, completing by time 8, giving $D_3 = 8$. Task 2 can never pre-empt task 3 more than once since task 3 finishes before task 2 can re-arrive (*i.e.* $D_3 \leq T_2$).

If task 3 were to take a little longer to complete (because, say, task 1 executes for an extra two ticks) then a first guess at D_3 would be $8 + 2 = 10$. However, now that task 3 is a little longer, task 2 can re-arrive and pre-empt task 3, giving a worst-case interference of 4. Figure 2 shows this situation.

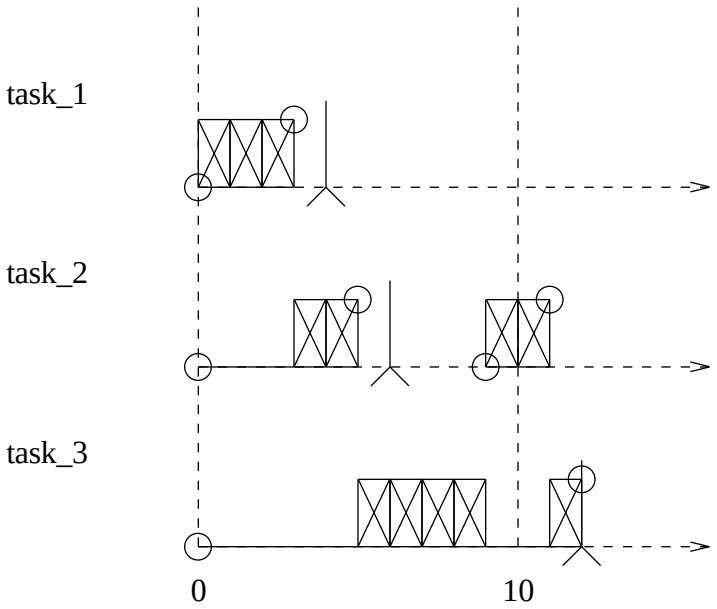


Figure 2

As can be seen from Figure 2 the worst-case response time of task 3 is now 12, just meeting the deadline.

In general, given prior knowledge of the worst-case response time D_i, the interference on task i from a task j is:

$$nC_j$$

Where n has a value such that $(n-1)T_j < D_i \leq nT$

Since $x = n$ when $n - 1 < \lceil x \rceil \leq n$ we can say that the worst-case interference from a task j on task i is given by:

$$\left\lceil \frac{D_i}{T_j} \right\rceil C_j$$

Note that this value for the maximal interference holds regardless of whether j is periodic or sporadic. This is an important result since it means that run-time techniques such as aperiodic servers [8] are not needed. In fact, a periodic task can be regarded as a sporadic task, released by a regular timing event.

The total interference I_i is therefore given by:

$$I_i = \sum_{\forall j \in hp(i)} \left\lceil \frac{D_i}{T_j} \right\rceil C_j \qquad (2)$$

However, when equations (1) and (2) are combined, the term D_i appears on both the left and the right hand sides of the equation. It is possible to iterate to find a value for D_i:

$$D_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{D_i^n}{T_j} \right\rceil C_j \qquad (3)$$

The iteration starts with $D_i^0 = 0$, and terminates when $D_i^{n+1} = D_i^n$. It can be shown that $D_i^{n+1} \geq D_i^n$ and so the iteration can be halted early if either $D_i^{n+1} > \mathcal{D}_i$ or if $D_i^{n+1} > T_i$. It can also be shown that the iteration is guaranteed to converge if the processor utilisation is ≤ 100% [5].

Note that so far in our analysis we have not mentioned how a priority ordering for a given task set is obtained. Using the Rate Monotonic policy (where $\mathcal{D} = T$) a priority ordering according to period is optimal.* For the Deadline Monotonic policy (where $\mathcal{D} \leq T$) a priority ordering according to deadline is optimal [10]. The Rate Monotonic priority ordering is therefore a special case of the Deadline Monotonic ordering. In general, however, no simple priority ordering policy may exist (we have already mentioned that for some tasks no value for $\mathcal{D}$ can be determined *a priori*, and hence the Deadline Monotonic policy cannot be used for those tasks). Indeed, in a distributed system with complex tradeoff finding an optimal priority ordering may be NP-Hard, and other sub-optimal techniques such as Simulated Annealing [6] are appropriate [17].

4. THE EXTERNAL BLOCKING PROBLEM

In this section we show how so-called *external blocking* causes problems with the analysis presented so far, and indicate how external blocking is often encountered in real systems. We then show how the analysis presented so far can be extended to allow for such blocking.

We define external blocking time as the time a task spends waiting for an external event to occur. If there are no lower priority tasks ready to execute when a task externally blocks then the processor is idle. This is the essential difference between external blocking and internal blocking (such as that encountered when locking semaphores according to the Priority Ceiling Protocol). With internal blocking the processor can not be idle — a higher priority task cannot be blocked unless a lower priority task is able to run.

The analysis presented in the previous section is not sufficient when tasks can experience external blocking. Consider the following task set, comprising of two tasks with $C_1 = 3$, $T_1 = 12$, $X_1 = 4$ (where X_1 is the maximum external blocking time), $C_2 = 6$, $T_2 = 20$, and $X_2 = 0$. Task 1 is of higher priority than task 2. In this example we are concerned with the schedulability of task 2: we have $\mathcal{D}_2 = 10$. Task 1 experiences the external block because it needs a message before it can commence. The message is guaranteed to arrive no later than 4 ticks after the release of task 1.

Using our current analysis we have:

$$D_2^0 = 0$$

$$D_2^1 = C_2 = \left\lceil \frac{D_2^0}{T_1} \right\rceil C_1 = 6$$

* 'optimal' in the sense that if a system is schedulable according to some other priority ordering then the Rate Monotonic ordering will also result in a schedulable system

$$D_2^2 = C_2 = \left\lceil \frac{D_2^1}{T_1} \right\rceil C_1 = 9$$

$$D_2^3 = C_2 = \left\lceil \frac{D_2^2}{T_1} \right\rceil C_1 = 9$$

The equation has converged, and hence $D_2 = 9$.

Since $D_2 \leq \mathcal{D}_2$ task 2 would be deemed schedulable. The following diagram shows a schedule for the two tasks when both are released together (the Liu and Layland worst-case).

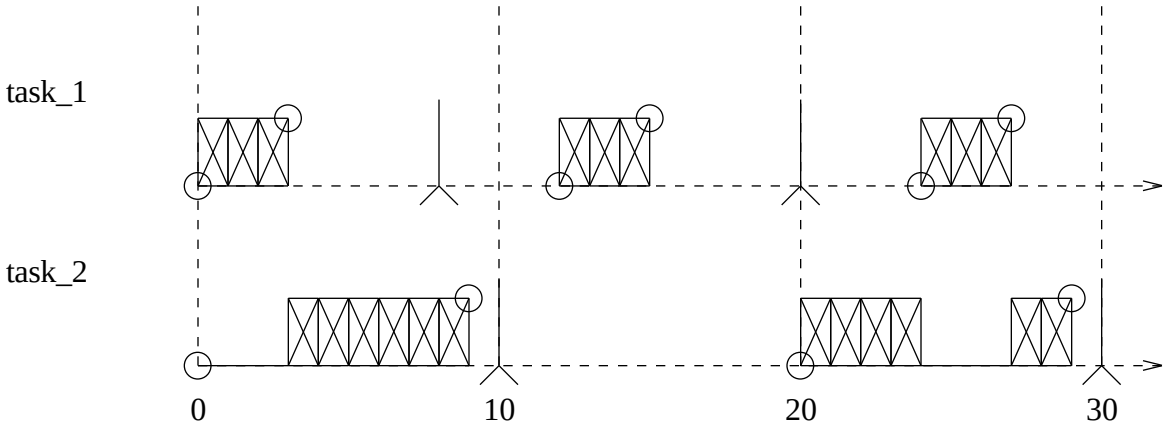


Figure 3

However, when external blocking is taken into account, task 2 will not always be schedulable. The following diagram shows this:

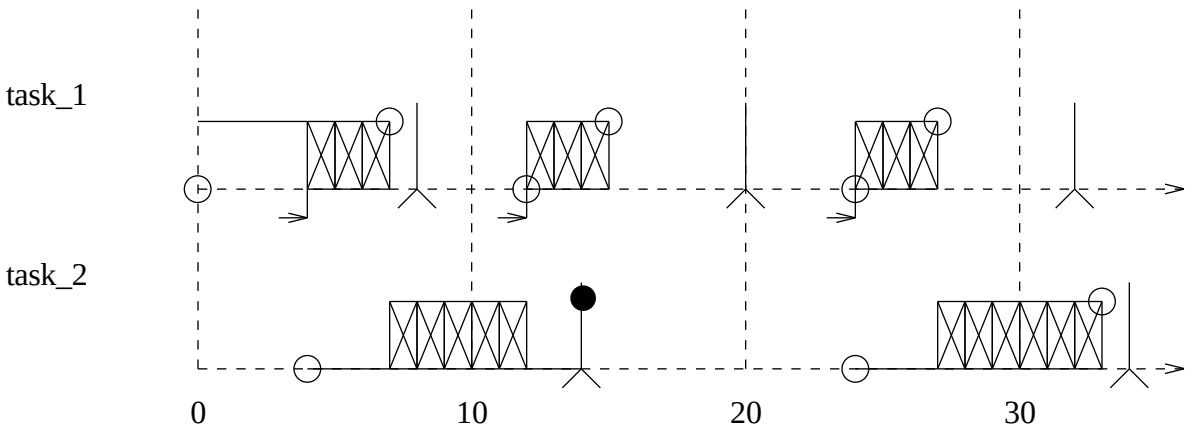


Figure 4

Task 1, although released at time zero, is suspended awaiting a message, which it receives at time 4. On the next release of task 1 the message is available and so the task can run immediately. Task 2 misses a deadline (indicated by the large black 'blob' in the STRESS diagram) because of external blocking. The reason is that the worst-case scheduling point no longer occurs at the Liu and Layland critical instant (where all tasks are released together), but at the point when task 2 is released at the same time as task 1

finishes blocking — task 1 can then effectively re-occur in a shorter time than the current analysis allows for, and so inflict a 'back to back hit'. This phenomenon is described by Rajkumar [15] with reference to external blocking when locking remote semaphores in a distributed system. This extra 'hit' can amount to at most an additional interference of C_1. The reason the current analysis fails is because the interference factor I_i is not sufficient. An upper bound on the interference to allow for the extra 'hit' might therefore be obtained, giving an interference factor of:

$$I_i = \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{D_i}{T_j} \right\rceil + 1 \right) C_j$$

$$\sum_{\forall j \in hp(i)} \left\lceil \frac{D_i + T_j}{T_j} \right\rceil C_j$$

In effect we are saying here that an extra 'hit' occurs if $D_i + T_j > T_j$. This is pessimistic, since the extra 'hit' is not bound to occur in all systems. Consider again Figure 4. If C_2 were 5 ticks then D_2 would be 8, and all computation for task 2 would be complete before task 1 re-arrived and pre-empted task 2. The extra 'hit' only occurs if $D_i + X_j > T_j$, and hence:

$$I_i = \sum_{\forall j \in hp(i)} \left\lceil \frac{D_i + X_j}{T_j} \right\rceil C_j \tag{4}$$

In Figure 4, D_2 would be 9 (according to Equation 3). Since $D_2 + X_1 > T_1$ (or $9 + 4 > 12$) task 2 gets an extra 'hit'. But if $C_2 = 5$ then D_2 would be 8, and since $8 + 4 \leq 12$, no extra 'hit' occurs.

Equation 3 can thus be modified to allow for external blocking:

$$D_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{D_i^n + X_j}{T_j} \right\rceil C_j \tag{5}$$

Note that the above equation allows semaphores to be locked and unlocked according to the Priority Ceiling Protocol. However, for the analysis of this protocol to be valid we assume that a task does not block externally after a semaphore lock has been obtained during the current invocation. This is a stronger assumption than simply disallowing an external block while holding a semaphore lock, as assumed by Rajkumar [15], since permitting a lower priority task to re-lock a semaphore while the higher priority task is externally blocked could result in the higher priority task being blocked more than once.

Having extended the scheduling analysis to handle external blocking it is useful to indicate how external blocking can occur in a system. So far we have described external blocking by assuming that a task voluntarily suspends itself on an external event (such as awaiting the arrival of a message from another processor). However, external blocking can sometimes occur in a system even if tasks do not voluntarily block awaiting an external event.

Precedence constrained subtasks statically allocated across multiple processors can be implemented using external blocking — a later subtask on one processor is blocked awaiting an indication of termination of all direct predecessors (perhaps by the arrival of a message from the network). The worst-case external blocking factor of such subtask can be computed by knowing the worst-case response times of predecessor subtasks located on other processors, and by knowing the worst-case communications delay. Because subtasks

never execute before being externally blocked awaiting the termination of predecessors they can not have attempted to lock a semaphore before the precedence blocking, and hence cannot have been internally blocked by a lower priority task. The exact details of how these external blocking factors are found, and how the schedulability of subtasks and other tasks in the processor are determined, are beyond the scope of this paper.

The implementation of a priority scheduler can also introduce external blocking. Consider a single processor where periodic and sporadic tasks are scheduled by a scheduler which is invoked by a periodic clock interrupt — so called 'tick driven scheduling'.

Assume the period of the scheduler is T_{tick}, and that the scheduler, once invoked, will take no more than C_{tick} computation time. Consider the following sequence of events: the scheduler is released at time $t = 0$ and examines an I/O register to see if the sporadic task s is to be released (in a real tick-driven system the scheduler might poll an I/O register for the condition for the release of s). The condition for the release is not true and the scheduler continues executing (ultimately terminating after taking time C_{tick}). Just after the time the scheduler has polled the I/O register the condition for the release of the sporadic becomes true. However, the sporadic cannot be released until the scheduler has been invoked at time $t = T_{tick}$. Hence the sporadic is effectively externally blocked for a maximum time T_{tick}, awaiting the timer which invokes the scheduler. The following STRESS diagram shows how a sporadic task is externally blocked by a tick-driven scheduler.

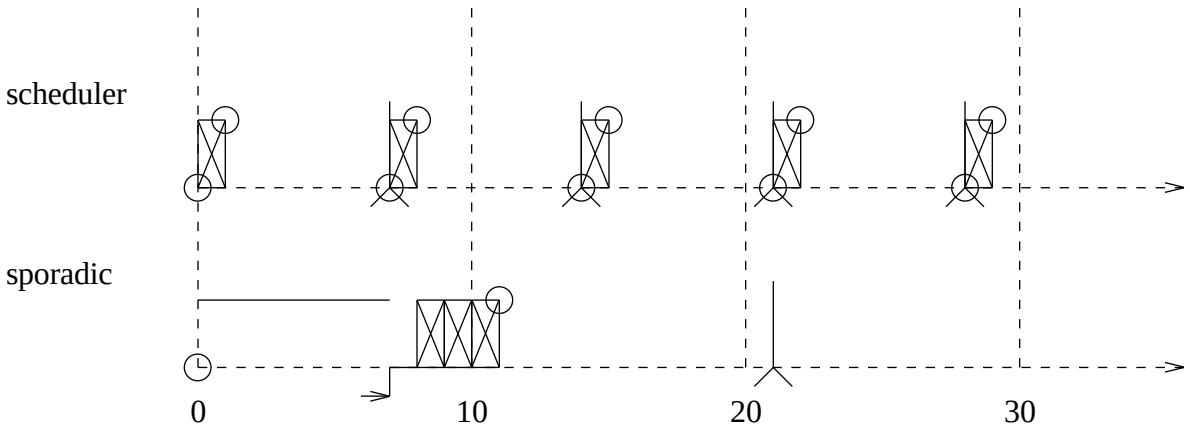


Figure 5

The tick-driven scheduler executes for $C_{tick} = 1$, with $T_{tick} = 7$. The worst-case execution time of the sporadic task is 3 time units. As can be seen, the sporadic task is externally blocked for 7 time units.

Tasks which always arrive as the scheduler is released do not experience external blocking — in the case study described later all periodic tasks have periods which are exact multiples of T_{tick}, with release times measured in scheduler ticks. Hence these tasks can be considered to always arrive as the scheduler is released.

5. DISCUSSION AND CASE STUDY

In this section we analyse and discuss the task set of a small avionics case study undertaken by Locke *et al* [12].

A number of mostly-periodic tasks implement an avionics weapons management subsystem. There is a single sporadic task, and a single task where the deadline of the task is less than the period (for 'jitter' requirements). Task priorities are assigned according to the Deadline Monotonic policy. Originally, the tasks were analysed according to the Rate Monotonic schedulability analysis derived by Sha *et al* [7]. The study reports that, using this analysis, out of a set of 18 tasks only the eight highest priority tasks could be guaranteed to meet their deadlines. In simulations nearly all tasks were found to meet their deadlines (two tasks are reported as missing their deadlines).

Equation (5) was applied to the task set described in [12], using the given priority assignment. For the single sporadic task an external blocking factor of 1000 μs was assumed (to account for the worst-case delay due to the operation of the tick-driven scheduler). The other tasks are all periodic with periods which are multiples of T_{tick} = 1000μs, and hence do not experience external blocking normally induced by tick-driven scheduling. Table 1 lists the tasks in priority order (task 1, the tick-driven scheduler, is the highest priority task), and gives the attributes and the derived response times of the tasks. All times given are in μs.

i	C_i	T_i	D_i	$\mathcal{D}_i$	B_i	X_i
1	51	1000	51	1000	0	0
2	3000	200000	3504	5000	300	0
3	2000	25000	5906	25000	600	0
4	5000	25000	11512	25000	900	0
5	1000	40000	13064	40000	1350	0
6	3000	50000	16217	50000	1350	0
7	5000	50000	20821	50000	750	0
8	8000	59000	36637	59000	750	0
9	9000	80000	47798	80000	1350	0
10	2000	80000	48949	80000	450	0
11	5000	100000	115966	100000	1050	0
12	1000	200000	137488	200000	450	1000
13	3000	200000	140641	200000	450	0
14	1000	200000	141692	200000	450	0
15	1000	200000	143694	200000	1350	0
16	3000	200000	145446	200000	0	0
17	1000	1000000	146497	1000000	0	0
18	1000	1000000	147548	1000000	0	0

Table 1
Task attributes and derived response times

As can be seen from the table, our analysis expects that all deadlines can be met except for task 11. Locke *et al* found that task 11 missed its deadline occasionally, and that task 16 missed a deadline once. Without more information we are unable to explain why task 16 missed a deadline. In general, however, our analysis agrees with the observed behaviour.

Moreover, it matches the observed behaviour more closely that the original rate monotonic analysis.

6. SUMMARY AND CONCLUSIONS

We have presented analysis in this paper which provides simple exact case analysis for systems scheduled at run-time with a static priority pre-emptive scheduler, allowing (with certain restrictions) tasks to suspend awaiting an external event — so-called *external blocking*. We have also shown how scheduling theory which allows for external blocking is required when a system using a tick-driven scheduler is analysed. A case study already presented elsewhere and analysed according to Rate Monotonic scheduling theory has been re-analysed using this scheduling theory.

The two most important aspects of the scheduling theory presented here are that the analysis demonstrates that older scheduling theory can be considered a special case of the analysis presented in this paper (systems previously analysed by the Rate Monotonic approach can now be re-analysed using more powerful techniques), and that the analysis presented here can be extended in a straightforward manner to allow more complex and powerful systems to be investigated.

7. REFERENCES

1. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ‘‘Hard Real-Time Scheduling: The Deadline Monotonic Approach’’, *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, USA (15-17 May 1991).
2. N. C. Audsley, ‘‘Resource Control For Hard Real-Time Systems: A Review’’, YCS 159, Department of Coputer Science, University of York (August 1991).
3. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ‘‘STRESS: A Simulator For Hard Real-Time System’’, RTRG/91/106, Real-Time Research Group, Department of Computer Science, University of York (October 1991).
4. T. P. Baker, ‘‘Stack-Based Scheduling of Realtime Processes’’, *Real Time Systems* **3**(1) (March 1991).
5. M. Joseph and P. Pandya, ‘‘Finding Response Times in a Real-Time System’’, *BCS Computer Journal*, pp. 390-395 (Vol. 29, No. 5, Oct 86).
6. S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, ‘‘Optimisation by Simulated Annealing’’, *Science*(220), pp. 671-680 (1983).
7. J. Lehoczky, L. Sha and Y. Ding, ‘‘The Rate Monotonic Scheduling Algorithm: Exact Characterisation and Average Case Behaviour’’, *Proceedings of the Real-Time Systems Symposium* (1989).
8. J. P. Lehoczky, L. Sha and J. K. Strosnider, *Enhancing Aperiodic Responsiveness in Hard Real-Time Environment*, Proceedings IEEE Real-Time Systems Symposium, San Jose, California, 1987.
9. J. P. Lehoczky, ‘‘Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines’’, *Proceedings 11th IEEE Real-Time Systems Symposium*, Lake Buena Vista, FL, USA, pp. 201-209 (5-7 Decmeber 1990).
10. J. Y. T. Leung and J. Whitehead, ‘‘On The Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks’’, *Performance Evaluation*, pp. 237-250 (Vol. 2, Part 4, Dec 1982).
11. C. L. Liu and J. W. Layland, ‘‘Scheduling Algorithms for Multiprogramming in a

Hard-Real-Time Environment'', *Journal of the ACM* **20**(1), pp. 46-61 (1973).

12. C. D. Locke, D. R. Vogel and T. J. Mesler, *Building a Predictable Avionics Platform in Ada: A Case Study*, Proceedings of the 12th Real Time Systems Symposium (December 1991).
13. Park, C.Y. and Shaw, A.C., ''Experiments with a program timing tool based on source-level timing schema'', *Computer* **24**(5), pp. 48-57, Computer (USA) (May 1991).
14. P. Puschner and C. Koza, ''Calculating the Maximum Execution Time of Real-Time Programs'', *Real Time Systems* **1**(1), pp. 159-176 (1989).
15. R. Rajkumar, ''Real-Time Synchronisation Protocols for Shared Memory Multiprocessors'', *Proceedings 10th IEEE International Conference on Distributed Computing Systems*, Paris, IEEE Computer Society Press (28 May - 1 June 1990).
16. L. Sha, R. Rajkumar and J. P. Lehoczky, ''Priority Inheritance Protocols: An Approach to Real-Time Synchronisation'', *IEEE Transactions on Computers* **39**(9), pp. 1175-1185 (September 1990).
17. K. Tindell, A. Burns and A. Wellings, ''Allocating Real-Time Tasks (An NP-Hard Problem made Easy)'', *Real Time Systems* (To Appear).

APPENDIX 1

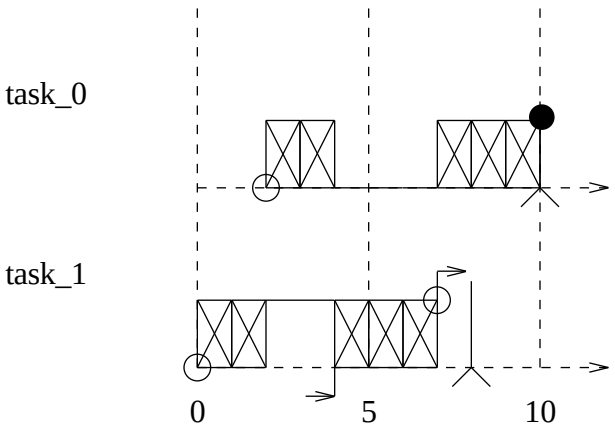
A Brief Description of *STRESS* Diagrams

STRESS diagrams illustrate the execution of tasks under the *STRESS* simulator. In these diagrams, time increases from left to right.

Task execution is represented by boxes. A task which is pre-empted is shown by a line at the level of the bottom of the boxes; a task which is blocked by a line at the level of the top of the boxes. These states are annotated by a variety of symbols.

Task release is marked by a low-level circle, and successful task completion by a high-level circle. If a task fails to meet its deadline, or otherwise fails to complete, then a filled high-level circle is used. Task deadlines are marked by a vertical line with a $\wedge$ mark at the bottom. Message transmission is marked by high-level outgoing arrow, and message reception by a low-level incoming arrow.

An example is shown below. Tasks task_0 and task_1 are released at times 2 and 0 respectively, and have deadlines at times 10 and 8 respectively; task_1 executes for two ticks, is externally blocked for two ticks, executes for three further ticks and then completes. It receives a message at tick 4, and sends a message at tick 7, as it completes. task_0 executes for two ticks, before being pre-empted at tick 4 and resumed at tick 7; it fails to meet its deadline and is killed.



Example Stress Diagram