

# A Simple Multi-Tasking Simulator

Adriaan de Beer<sup>1</sup> Colin Fidge<sup>2</sup>

<sup>1</sup>School of Information Technology & Electrical Engineering

<sup>2</sup>Software Verification Research Centre  
The University of Queensland, Australia

## Abstract

*The dynamic behaviour of multi-tasking systems is complex and hard to understand. We explain how a simple multi-tasking simulator was constructed by implementing scheduling theory's computational model in a commercial simulation toolkit. The resulting simulator displays multi-tasking behaviours graphically.*

## 1 Introduction

The widely-publicised task scheduling problems encountered during the Mars Pathfinder mission [3] remind us that modern embedded systems rely heavily on multi-tasking software. Programmers of avionics, aerospace, process control, and military software must therefore have a thorough understanding of the way their system will behave at run time. Such systems are subject to rigid real-time requirements, imposed by their operating environment, and comprise numerous concurrent tasks, in order to respond to simultaneous events. Unfortunately, the resulting systems are complex and hard to understand. Although formal theories can be used to prove timing properties of a multi-tasking program, they do not help the programmer to visualise its behaviour.

Our goal was to develop a tool that can display the behaviour of a multi-tasking system in an easily digested form. In particular, we wanted an educational aid that shows significant features of multi-tasking designs, such as the impact of task offsets or the way low-priority tasks can sometimes impede higher-priority ones. We also wanted source-level access to the tool, so that we could modify and extend it ourselves, but did not want to face a major programming challenge each time a change was made.

In this paper we explain how such a simulator was quickly and easily constructed using a commercial design and analysis toolkit. This was done by encoding scheduling theory's computational model as a system design in the toolkit. This allowed us to exploit the existing graph-

ical user interface and simulation capabilities of the toolkit and focus instead on significant scheduling theory concepts. Through examples, we also show how the resulting simulator's output confirms the predictions of formal schedulability analysis.

## 2 A Review of Scheduling Theory

Fixed-priority scheduling theory is founded on a particular 'computational model' of a multi-tasking system [1]. In this model, there is a static set of concurrent *tasks*, all residing on the same processor. A run-time *scheduler* allocates processing time to the tasks in discrete quanta. Each task has a static *base priority*, although the task may temporarily acquire a higher *active priority*. At run time, each task requires an infinite number of *invocations*. The time at which a task invocation becomes ready to execute is its *arrival* time. The time at which a task invocation actually begins execution is its *starting* time. This may be significantly later than the arrival time if, for instance, a higher-priority task was already running when the invocation arrived. The time at which a task invocation completes execution is its *finishing* time. Invocations of a task  $i$  are usually assumed to arrive regularly with a fixed *period*,  $T_i$ . (*Sporadic*, or non-periodic, tasks, arrive at irregular intervals but with a known minimum separation. The theory treats these as periodic by using their minimum separation time as the period.) For each task  $i$  the programmer specifies a *deadline*,  $D_i$ , by which each of its invocations must finish, measured relative to the invocation's arrival time. To support analysis, the programmer also postulates a worst-case *computation time*,  $C_i$ , for each invocation of task  $i$ . It is assumed that  $C_i$  includes the context-switching overheads associated with scheduling the task invocation—the run-time scheduler thus appears to operate instantaneously in the model. The worst-case difference between task  $i$ 's arrival and finishing times is its *response time*,  $R_i$ . Each task normally starts to perform invocations from time 0, but this can be delayed by assuming an initial *offset*,  $O_i$  [9].

In the model, tasks interact in two ways. High-priority

tasks can *preempt* an invocation of task  $i$ , and thus slow task  $i$ 's progress by up to its worst-case *interference* time,  $I_i$ . Communication between tasks is via mutually-exclusive access to *shared resources*, such as common memory or data buses. If a low-priority task has locked a shared resource, it can slow progress of an invocation of task  $i$  by up to its worst-case *blocking time*,  $B_i$  [10].

The theory then provides specific *schedulability tests* for particular scheduling policies and locking protocols. These allow programmers to predict whether a multi-tasking design will meet its deadlines or not. Early work was limited to 'rate monotonic' task priorities, and used a notion of 'processor utilisation' to assess schedulability. Recent work calculates each task's worst-case response time, and applies to any fixed-priority scheduling policy [5, 1, 10].

In particular, a set of tasks using a preemptive scheduling policy is schedulable if the following property holds for every task  $i$ . Assume that tasks have unique priorities and let  $hp(i)$  be the set of tasks with higher base priority than task  $i$ . For some number  $x$ , let  $\lceil x \rceil$  be the smallest integer greater than or equal to  $x$ , i.e.,  $x$  rounded up to the next whole number.

$$\begin{aligned} \text{Check } R_i &\leq D_i \\ \text{where } R_i &= C_i + B_i + I_i \\ \text{and } I_i &= \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \end{aligned}$$

In other words, task  $i$ 's response time  $R_i$  must not exceed its deadline  $D_i$ . Worst-case response time  $R_i$  is the sum of task  $i$ 's worst-case computation time  $C_i$ , its worst-case blocking time  $B_i$  (due to lower-priority tasks), plus its worst-case interference  $I_i$  (due to higher-priority tasks). Task  $i$ 's interference  $I_i$  is the sum, for each higher-priority task  $j$ , of task  $j$ 's computation time  $C_j$  times the number of arrivals of task  $j$  in an interval of duration  $R_i$ . The worst-case number of arrivals is found by dividing  $R_i$  by task  $j$ 's period  $T_j$  and rounding up. Thus the equation defining response time is recursive—term  $R_i$  appears on both sides of the equality—but it can be solved iteratively [1].

### 3 The Foresight Toolkit

Foresight<sup>1</sup> is an integrated toolkit for the design and analysis of complex systems [7]. It includes an extensive range of tools for constructing executable models, simulating and analysing their behaviour, and producing prototype implementations from them. It also provides predefined libraries of system components. For our purposes we needed only a small portion of the overall toolkit. Our computational model was constructed using Foresight's *Model Editor* as a set of Data Flow Diagrams, with new components

defined in Foresight's Mini-Spec programming language. The model was checked for syntactic and static semantic correctness using Foresight's *Model Analyzer*, and then executed using its interactive *Simulator* tool.

At the top level, Foresight models are constructed as *Data Flow Diagrams*. These consist of computational *process* elements connected by *flow arcs* that carry data *tokens*. The process elements themselves may be composed of further Data Flow Diagrams, State Transition Diagrams, predefined Library elements, or Mini-Spec procedures. Data flow input connectors may be either *discrete*, in which case they offer one value for each token sent along the arc, or *continuous*, in which case they always offer the most recently sent value.

The *Mini-Spec* language is a simple programming language for defining processes that manipulate data flows. The Mini-Spec notation includes: a *declaration* part for input and output flows, and local variables and constants; *initialisation* code; and a *procedure* to be performed each time the process fires. A Mini-Spec process *fires* when tokens are available on all of its input flows. (If all of its inputs are continuous, then it fires whenever an input value changes.)

Importantly for our purposes, Foresight gives the programmer explicit control over the passage of simulation time. By default, the execution time of process elements and the propagation delay of data tokens is assumed to be zero. Thus, although events in the simulation always occur in a well-defined sequence, 'time' does not progress unless this is explicitly built into the model. To do so, predefined processes are provided for introducing propagation delays into flows, and the Mini-Spec language includes a **delay** statement which can be used to introduce computational delays into processes.

### 4 Previous Work

The idea of graphically displaying the behaviour of multi-tasking systems is by no means new. For instance, commercial debugging tools such as WindView [11] allow a suitably instrumented operating system to log run-time events, and display the behaviour of an operational multi-tasking system as a trace over time.

More relevant to our goals are simulators that allow the behaviour of a planned multi-tasking system to be predicted. We were particularly inspired by the STRESS tool which simulates the behaviour of hard real-time systems under a preemptive scheduling policy [2]. Programmers using STRESS can express the design of a multi-tasking system in a modelling language which supports typical multi-tasking concepts such as tasks, shared objects and semaphores. The model can then be simulated, with the results shown graphically. For each task invocation, its arrival and finishing times are displayed, as well as its deadline and the intervals

<sup>1</sup>Foresight is a trademark of Nu Thena Systems.

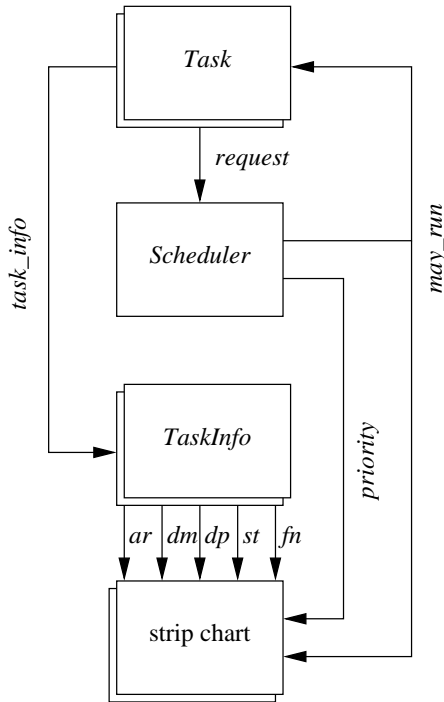


Figure 1. Simulator Data Flow Diagram

during which the task ran.

Similarly, the DRTSS simulator, which forms part of the PERTS prototyping environment, can simulate distributed real-time systems [6]. A real-time system is modelled in PERTS as a set of physical components, including processors which may support different scheduling policies, and a task set, defined via two sorts of graphs. A ‘task graph’ defines each task’s parameters and the precedence constraints between task invocations. A ‘resource graph’ defines the relationships between, and access constraints on, shared resources. The simulation then generates a list of primitive events which can be displayed graphically using a separate postprocessing tool.

Most recently, Palopoli et al. described a toolkit for simulating both the functional and timing behaviour of multi-tasking systems [8]. They build system models as data-flow designs using a collection of C++ libraries. Predefined objects in the libraries model typical system components such as network links, processors, memory buffers, etc. The simulator is especially targetted at process control applications.

Our simulator has much in common with all these tools. Our work differs, however, in our desire to include a non-preemptive scheduling policy, which was considered important because the cheapness and predictability of this approach is felt to be attractive to embedded systems programmers [4]. Also, to help understand scheduling theory principles, we wanted a tool that displays not only which task

is running, but also the active priorities of tasks, as this is significant to understanding how manipulation of priorities achieves mutual exclusion in ‘priority inheritance’ protocols [10].

## 5 The Multi-Tasking Simulator

Our challenge, therefore, was to represent scheduling theory’s computational model of priority-allocated processing quanta in the Foresight toolkit’s data flow-based design language. Figure 1 shows the resulting model.

One or more *Task* processes model application-specific concurrent tasks and are parameterised by the task’s characteristics such as its period, deadline, priority, etc. The *Scheduler* process controls the execution ordering of the tasks according to a specific scheduling policy. Two ‘strip chart’ processes display which task is running, and the active priority level of the currently running task, respectively. Discrete data flows connect the tasks to the scheduler, the scheduler to the tasks, and both the tasks and scheduler to the strip charts. The *TaskInfo* processes reformat data tokens from the tasks to make them suitable for display. Each of the *Task*, *Scheduler* and *TaskInfo* processes is implemented by a small Mini-Spec procedure.

Given this design, with a separate simulation process for each task, the first issue to be solved was how to coordinate the tasks so that only one makes progress at a time. In the simulator, all processes are capable of executing simultaneously, but for our scheduling model we need the tasks to interleave their actions. To achieve this, the *Task* processes themselves keep track of the current time and, using their *period*, *offset* and *comp.time* parameters, decide whether they wish to execute or not at each step. They then send a *request* token consisting of either the active priority at which the task wishes to execute, or 0 if the task does not need to execute at the current time, to the *Scheduler* process.

The *Scheduler* process uses the priorities in the requests to decide which task may make progress. (In effect, the set of *request* tokens models the ‘ready queue’ in an actual scheduler.) The *Scheduler* process then returns a *may\_run* token containing the number of the task which it has selected to run at this step. The corresponding *Task* process uses this knowledge to decrement a local variable which tracks the computation time remaining in its current invocation. Other, ‘non-running’ *Task* processes leave their state unchanged.

Scheduling theory assumes that the available processing time is divided into discrete units. The *Task* and *Scheduler* processes therefore run in lockstep. Each of their procedures reads its inputs, calculates results, and produces the corresponding outputs, instantaneously in simulation time. They then all end by performing a **delay** of one time unit.

Different scheduling policies can be programmed into

- Arrival time
- + Starting time
- × Finishing time
- Deadline: met
- Deadline: missed!

**Table 1. Symbols used in traces**

the *Scheduler* process. We implemented fixed-priority preemptive and non-preemptive scheduling, with the choice changed by a flag inside the *Scheduler* element. In both cases the scheduler remembers the task that ran at the last step, if any. In the preemptive case, the *Scheduler* procedure examines the set of *request* tokens and chooses the task with highest active priority to be the one that may run. However, if several waiting tasks have equal highest priority, the scheduler always chooses the one that ran in the last step, if any. This ensures that there is no unnecessary preemption—a task can be preempted by another task with strictly higher priority only. In the non-preemptive case, the task that ran at the last step is allowed to run for as long as it requests time. Otherwise, the task with highest priority is chosen.

So far we have said nothing about how tasks communicate. As noted above, scheduling theory assumes that tasks interact via shared resources, but the design in Figure 1 does not include ‘shared resource’ processes or any other form of direct communication between the tasks. This was not necessary because we assumed the use of a *priority ceiling* locking protocol [10] to achieve mutual exclusion. In this approach, a task that accesses a shared resource raises its active priority to the highest base priority of any task that *may* use the resource. This prevents other tasks that could access the resource from preempting the task that is using it. There is thus no need for a separate locking mechanism. Our model takes full advantage of this approach. When we wish to simulate the effect of a task locking a shared resource, we have the *Task* process send the corresponding ‘ceiling’ priority in its *request* token. If the scheduler allows the task to run at this priority then it effectively has the lock. There is thus no need for communicating *Task* processes to interact directly in our model—synchronisation between tasks competing for shared resources occurs indirectly, via the *Scheduler* process.

Since the pattern of shared resource usage may be quite complex for a particular task invocation, it would have been impractical to ‘hardwire’ the behaviour of the active priorities into the *Task* processes. Therefore, *Task* processes can read a script from a text file (*res.file*) which defines the active priorities they should request at each step of their execution [6]. (An example is shown in Section 7.)

Finally, we must display the results of the simulation in an easily understood form. A ‘running task’ strip chart shows the base priority of the currently running task, as de-

Task no.	Base priority	Period $T$	Comp. time $C$	Dead-line $D$	Offset $O$
1	1 (low)	10	4	8	0
2	2 (med)	8	3	4	2
3	3 (high)	7	2	3	1

**Table 2. Task set used in Section 6**

termined from the *may-run* tokens produced by the scheduler. (We assume here that task numbers and base priorities are the same.) As already explained, the *Task* processes keep track of the arrival, starting and finishing times for each of their invocations. By also having these processes note whether the invocation meets its deadline or not, we have all the information needed for display. Each *Task* process packages this data and sends it to the *task.info* flow. The *TaskInfo* processes then separate the arrival time (*ar*), starting time (*st*), finishing time (*fn*), deadline met (*dm*) and deadline passed (*dp*) fields and send them to the running task strip chart where each event is shown as a different symbol. (The *TaskInfo* processes are not essential but reduce the clutter in the Data Flow Diagram when there are many tasks.) The various symbols used appear in Table 1.

However, showing which task is currently running is not sufficient to fully understand how a multi-tasking computation works. The way tasks change their active priorities is crucial to understanding how low-priority tasks may block higher-priority ones. We therefore also have the *Scheduler* process produce a separate (active) *priority* flow which is displayed in a second strip chart. This makes it possible to see the difference between the running task’s base and activity priorities, and hence determine whether it is accessing any shared resources used by higher-priority tasks.

Implementing all of these features in Foresight was straightforward. Ignoring declarations, the Mini-Spec procedures for the *Task*, *Scheduler* and *TaskInfo* processes consisted of only 71, 45 and 22 lines of code, respectively (and most of this was associated with initialisation and formatting the output, rather than computation). The only significant difficulty encountered was achieving a good understanding of Foresight’s data flow firing rules, and how they interact with the Mini-Spec code.

## 6 Example: Preemptive Vs Non-Preemptive Scheduling

Consider a set of three independent tasks with characteristics as shown in Table 2. (Since there is no communication between the tasks, their blocking times  $B$  are all zero.) For instance, task 3 has the highest priority and needs up to 2 seconds of computation time every 7 seconds. Each of

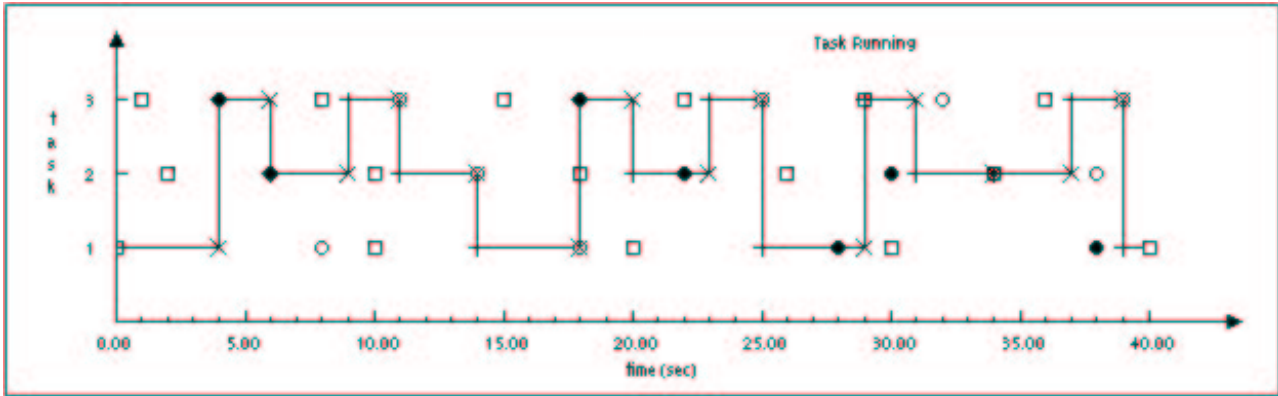


Figure 2. Non-preemptive simulation of the task set in Table 2

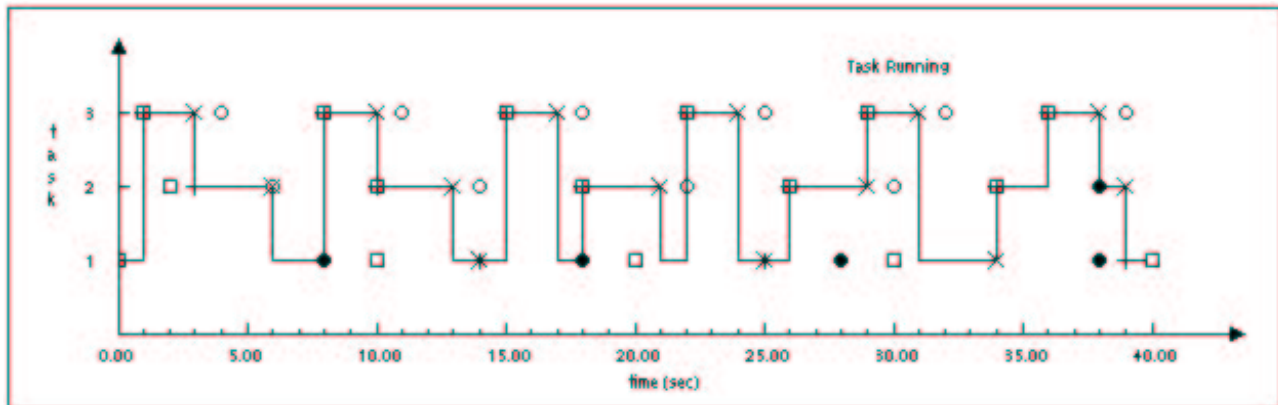


Figure 3. Preemptive simulation of the task set in Table 2

task 3’s invocations is required to finish within 3 seconds of its arrival. An initial offset means that task 3’s first invocation arrives at time 1, the second invocation at time 8, and so on. Similarly for the other two tasks.

Figure 2 shows the simulator’s ‘task running’ trace for this task set, assuming a *nonpreemptive* scheduling policy. Since it has no initial offset, lowest-priority task 1 arrives at time 0 and begins executing straight away. Although invocations of higher-priority tasks 3 and 2 arrive at times 1 and 2, respectively, the non-preemptive scheduling policy means that they must wait until task 1 finishes. When it does so, at time 4, the highest-priority waiting task, number 3, begins executing. However, this proves to be too late, as task 3’s first deadline was at time 4 (3 seconds after its arrival) and the deadline has already expired, as shown by the missed deadline symbol ‘•’. We thus immediately see the major disadvantage of non-preemptive scheduling: there is no way to prevent low-priority tasks from causing high-priority ones to fail. The second invocation of task 3 fares better and meets its deadline at time 11. Nevertheless, the scattering of missed deadline symbols in Figure 2 re-

veals that all three tasks fail to meet some of their deadlines under a non-preemptive scheduling policy.

Figure 3 then shows the simulation for the same task set, but with a *preemptive* scheduling policy. The simulation accords precisely with the results of formal schedulability analysis for preemptive scheduling. Ignoring the effects of the initial offsets, we can calculate the response times for the three tasks as follows. (The schedulability test used in this article does not account for offsets—analysis in the presence of offsets is surprisingly complex [9].) Since there are no tasks  $j$  with priority higher than task 3, its response time is trivial.

$$R_3 = C_3 + B_3 = 2$$

As this is less than its deadline of 3, we can conclude that task 3 will always meet its deadline. Figure 3 confirms this. Whenever task 3 arrives it always starts executing immediately, meeting its deadline with 1 second to spare.

Next we calculate the worst-case response time for

Task no.	Base priority	Period $T$	Comp. time $C$	Dead-line $D$	Block. time $B$
1	1 (low)	14	4	12	0
2	2 (med)	12	4	6	2
3	3 (high)	8	2	4	2

**Table 3. Task set used in Section 7**

Time interval	Active priority
0–1	1 (base)
1–3	3 (high)
3–4	1 (base)

**Table 4. Computation profile of task 1, Table 3**

medium priority task 2.

$$R_2 = C_2 + B_2 + \left\lceil \frac{R_2}{T_3} \right\rceil \cdot C_3 = 5$$

The recursive equation converges with a value of 5. However, since this exceeds task 2’s deadline of 4, we conclude that task 2 is *not* schedulable. Although the first four invocations of task 2 in Figure 3 succeed, this worst-case response time can be seen to occur on the fifth invocation which arrives at time 34. This invocation is preempted by task 3 at time 36 and therefore does not finish until time 39, 5 seconds after arrival, exactly as predicted. (If we had set all offsets to 0, task 2 would have missed its deadline at the first invocation. By choosing the particular offsets shown in Table 2 we delayed the point at which task 2 first missed a deadline, but did not make it schedulable.)

Given that task 2 is not schedulable, it is perhaps not surprising to learn that lower-priority task 1 is also unschedulable.

$$R_1 = C_1 + B_1 + \left\lceil \frac{R_1}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_1}{T_3} \right\rceil \cdot C_3 = 14$$

Thus the predicted worst case response time for task 1 not only exceeds its deadline of 8, but even exceeds its period of 10. Therefore, two invocations of the task can be waiting to execute at the same time. This situation can be seen to occur in Figure 3. The first invocation of task 1 arrives at time 0 but, due to interference from tasks 2 and 3, does not finish until time 14. However, the second invocation has already arrived by then, at time 10. The second invocation thus starts as soon as the first finishes, and task 1 continues running beyond time 14 (but only until time 15, when it is again preempted).

Overall, comparing the pattern of missed deadlines in Figures 2 and 3 clearly highlights the advantage of preemptive

over non-preemptive scheduling. Preemptive scheduling allows higher-priority, and presumably more important, tasks to meet their deadlines more often than non-preemptive scheduling.

## 7 Example: Communicating Tasks

Now consider a set of three communicating tasks as shown in Table 3. (Offsets  $O$  are all assumed to be 0.) We assume that tasks 3 and 1 interact by both accessing the same shared resource. Mutual exclusion is supported by using the priority ceiling protocol. To calculate blocking time  $B$  for each task in this situation we need to know the worst-case time for which other tasks may access the resource [10].

Each invocation of low-priority task 1 has a computation time of 4 seconds. In this period, we define it to behave as shown in Table 4. For the first second it runs at its base priority. After this it accesses the shared resource for 2 seconds. Since this resource is shared with high-priority task 3, the ceiling locking protocol raises task 1’s active priority to level 3 while the resource is being used. In its fourth second task 1 again runs at its base priority. (There is no need to specify such a profile for task 3 because it does not raise its priority above its own base level of 3 when it accesses the resource.)

Given the knowledge that task 1 may access the shared resource for up to 2 seconds, scheduling theory tells us that we can determine the blocking times shown in Table 3 [10]. Task 1 itself cannot be blocked because there are no tasks with lower priority. Task 3 can be blocked for up to 2 seconds because it may find that the shared resource is already being used by task 1. Surprisingly, however, task 2 also may be blocked for up to 2 seconds, even though it does not use the shared resource. This occurs because when task 1 uses the resource, its active priority is raised to level 3, and it can thus ‘block’ (preempt) task 2.

Once again, the behaviour predicted by scheduling theory for this task set is confirmed by our simulator, as shown in Figure 4. Formally, we can calculate the response times for the three tasks as follows. Since there are no tasks with priority higher than task 3, its response time is again straightforward.

$$R_3 = C_3 + B_3 = 4$$

Since this equals its deadline, we can conclude that task 3 is schedulable, despite potential blocking from a lower priority task. Figure 4 shows this happening. For instance, the first invocation of low-priority task 1 starts executing at time 6. As per its computation profile in Table 4, task 1 accesses the resource it shares with task 3 at time 7. This can be seen in the ‘priority’ trace—at time 7 the active priority of the running task increases from 1 to 3, even though

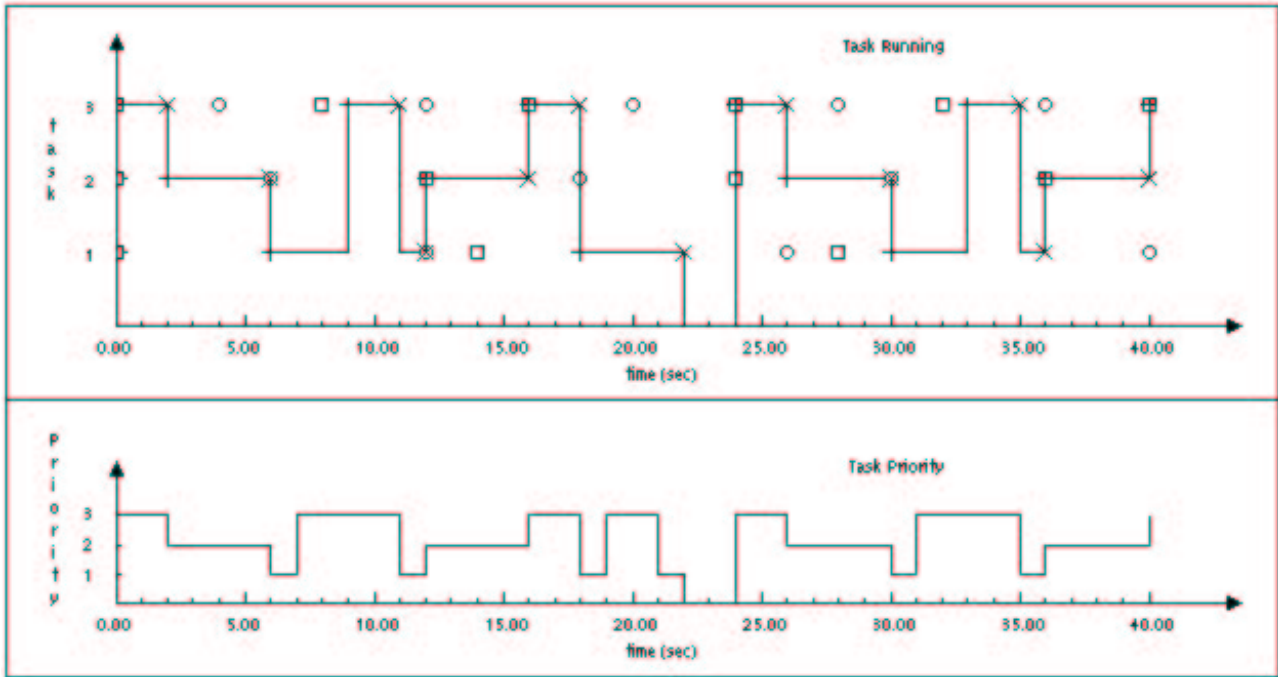


Figure 4. Preemptive simulation of the task set in Table 3, from times 0 to 40

the ‘task’ trace reveals that low-priority task 1 is still running. The second invocation of task 3 then arrives at time 8. Even though task 3 has the highest base priority, it cannot begin executing until time 9, after task 1 has stopped using the resource. Thus, even though we are using a preemptive scheduling policy, low-priority task 1 can still slow the progress of high-priority task 3 by accessing the resource they share.

We calculate the response time for medium-priority task 2 as follows.

$$R_2 = C_2 + B_2 + \left\lceil \frac{R_2}{T_3} \right\rceil \cdot C_3 = 8$$

Since this exceeds task 2’s deadline of 6, we conclude that task 2 is not schedulable. However, the simulation in Figure 4 does *not* seem to confirm this—each invocation of task 2 finishes before its deadline. Therefore, we allowed the simulation to continue for a further 40 seconds of simulation time, as shown in Figure 5. Now we finally see an invocation of task 2 miss its deadline, as predicted. Task 2 arrives at time 60, but cannot start executing until time 61 because low-priority task 1 is blocking it by accessing the shared resource. The priority graph reveals that task 1 started executing at active priority 3 at time 59. This is an example of the indirect blocking effect described above: task 2 is blocked by task 1 even though it does not use the shared resource. At time 61 task 1 stops using the resource and reverts to its base priority, and task 2 starts executing.

However, it is preempted by task 3 at time 64, and does not regain control of the processor until time 66 by which time it has already missed its deadline. The invocation of task 2 finally finishes at time 67, 7 seconds after it arrived.

The next invocation of task 2, which arrives at time 72, also misses its deadline under similar circumstances. We can now clearly see why it took so long for the simulation to produce a situation where task 2 fails. This outcome requires a combination of blocking from lower-priority task 1 and interference from higher-priority task 3. Neither task on its own can cause task 2 to fail.

Since task 2 was not schedulable, we may expect that lower-priority task 1 is also unschedulable, but this does not prove to be so.

$$R_1 = C_1 + B_1 + \left\lceil \frac{R_1}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_1}{T_3} \right\rceil \cdot C_3 = 12$$

Thus the worst case response time for task 1 equals its deadline. Figures 4 and 5 support this. All invocations of task 1 meet its deadline. The worst cases are the first invocation, which finishes at time 12, and the fifth invocation, which finishes at time 68, both exactly at the deadline.

## 8 Conclusion

We have seen how a useful multi-tasking simulator was quickly and easily constructed merely by encoding scheduling theory’s computational model in a general-purpose sim-

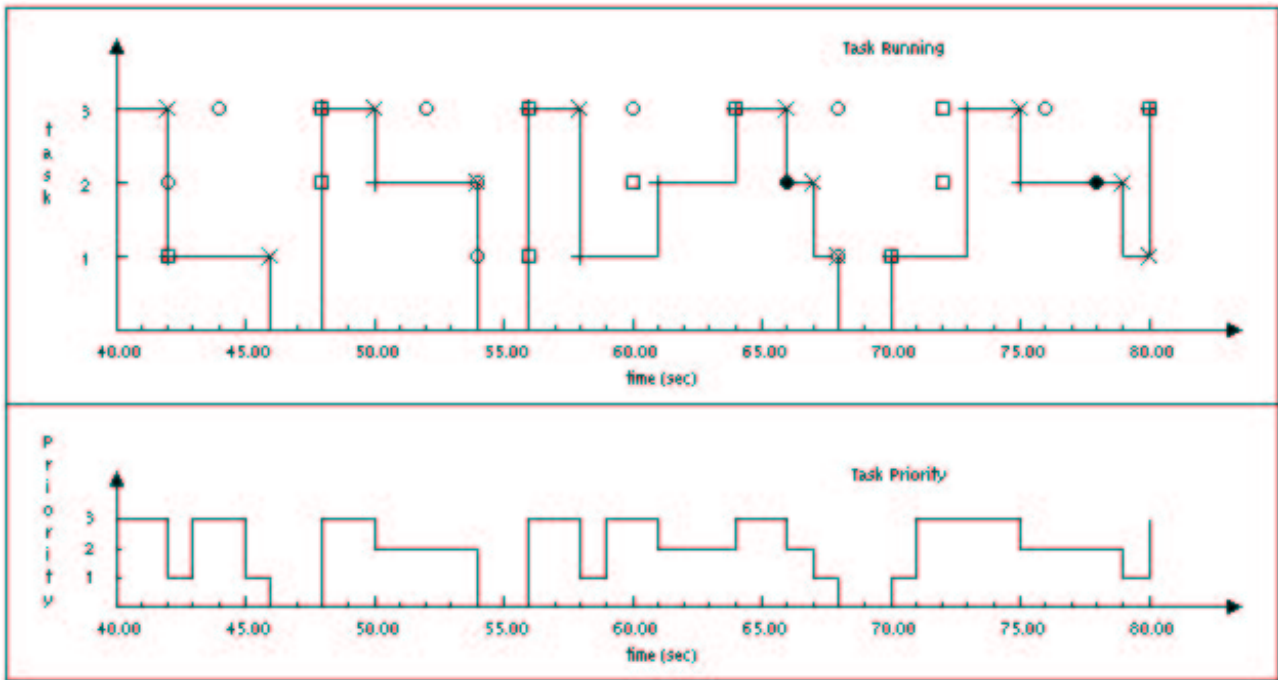


Figure 5. Preemptive simulation of the task set in Table 3, from times 40 to 80

ulation toolkit. This is a testament to both the elegance of the theory [1] and the flexibility of the toolkit [7]. Of particular interest was the way in which competition for shared resources could be simulated without the need to introduce any direct communication between tasks.

There are many ways the tool could be extended. Other scheduling policies and locking protocols could be simulated, and overheads associated with the scheduler could be made explicit, rather than incorporated into the task's computation times. More significantly, a multi-processor simulation could be developed by including multiple *Scheduler* components.

**Acknowledgements** We wish to thank Nu-Thena Systems and Don Williams of Bogong Technologies for making the Foresight tool available for this project. This research was funded in part by ARC Large Grant A49702415, *Efficient Development of Verified Concurrent Real-Time Programs Through Tool Support*.

## References

- [1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, Sept. 1993.
- [2] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. STRESS: A simulator for hard real-time sys-

tems. *Software—Practice & Experience*, 24(6):543–564, June 1994.

- [3] L. P. Briand and D. M. Roy. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. IEEE Computer Society Press, 1999.
- [4] A. Burns and A. J. Wellings. Simple Ada 95 tasking models for high integrity applications. Department of Computer Science, University of York, May 1996.
- [5] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.
- [6] J. W. S. Liu, C. L. Liu, Z. Deng, T. S. Tia, J. Sun, M. Storch, D. Hull, J. L. Redondo, R. Bettati, and A. Silberman. PERTS: A prototyping environment for real-time systems. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):161–177, 1996.
- [7] Nu-Thena Systems. *Foresight User's Guide*, Sept. 1996. Release 4.10.
- [8] L. Palopoli, G. Lipari, G. Lamastra, L. Abeni, G. Bolognini, and P. Ancilotti. An object-oriented tool for simulating distributed real-time control systems. *Software—Practice & Experience*, 32(9):907–932, July 2002.
- [9] K. Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS 221, Department of Computer Science, University of York, 1994.
- [10] K. Tindell. Deadline monotonic analysis. *Embedded Systems Programming*, 13(6):20–38, June 2000.
- [11] D. Wilner. Windpower, a new vision of real-time embedded software development. *Real-Time Magazine*, 93/4:56–61, 1993.