Another problem in modeling computer systems is that of calibrating and validating the model against the real world. With a performance model of an existing computer system, the calibration and validation process uses data obtained by measuring the performance of the real system. The calibration and validation process is usually an iterative statistical procedure which attempts to ensure that the predictions of the model are not significantly different from the real world performance [Beilner72, Gomaa78]. Once this has been achieved, the model may be used for estimating the impact of changes to the system, by playing "what if" scenarios.

The calibration and validation process for a performance model of a system that does not yet exist is much more error-prone. Certain parameters, such as operating system overhead (e.g., for context switching and servicing task communication requests), can be obtained by performance measurement of the actual target system. However, many other parameters, such as task execution times, have to be estimated. Thus the accuracy of the model predictions are highly dependent on the accuracy of the estimates of the performance parameters.

11.3 Petri Nets

Finite state machines have been used to model systems in which the sequence of events plays an important role; that is, the response to a given input event depends not only on the type of event but also on what has happened previously in the system. However, finite state machines are strictly sequential and thus they cannot be used to model parallelism.

An alternative modeling approach to finite state machines is that of Petri nets [Peterson81]. Petri nets can model concurrent systems directly, so they are a more powerful tool than finite state machines. Indeed, the finite state machine is a sequential subset of the Petri net.

A Petri net is represented as a directed graph. The two types of nodes supported are called places (depicted by circles) and transitions (depicted by bars). The execution of a Petri net is controlled by the position and movement of markers called tokens. Tokens are depicted by black dots that reside in the places of the net. Tokens are moved by the firing of the transitions of the net. A transition is enabled to fire when all its input places have a token in them. When the transition fires, a token is removed from each input place and a token is placed on each output place.

Various extensions have been proposed to Petri net models. In particular, Timed Petri nets are particularly useful for modeling real-time systems. Whereas a basic Petri net has no times associated with it, Timed Petri nets allow finite times to be associated with the firing of transitions [Coolahan83], allowing a Petri net model to be analyzed from a performance perspective.

Petri nets can be used as analysis tools as well as modeling tools. Petri nets have been used successfully to model hardware systems, communication protocols, and software systems. In the latter case, Petri nets have been used to model task synchronization [Peterson81], message communication between tasks [Peterson81], and Ada concurrent tasking applications.

The analysis capability of Petri nets is very powerful; for example, they can be used to detect reachability and deadlocks. Furthermore, a stochastic Petri net model can be used to analyze throughput of the proposed system. Thus Petri nets are an attractive proposition for real-time and distributed systems, where throughput and response times are important considerations.

11.4 Real-time Scheduling Theory

11.4.1 Introduction

Real-time scheduling theory addresses the issues of priority-based scheduling of concurrent tasks with hard deadlines. The theory addresses how to determine whether a group of tasks, whose individual CPU utilization is known, will meet their deadlines. The theory assumes a priority pre-emption scheduling algorithm, as described in Chapter 3. This section is based on the reports on real-time scheduling produced at the Software Engineering Institute [Sha90, SEI93], which should be referenced for more information on this topic.

As real-time scheduling theory has evolved, it has gradually been applied to more complicated scheduling problems. Problems that have been addressed include scheduling independent periodic tasks, scheduling in situations where there are both periodic and aperiodic (asynchronous) tasks, scheduling in cases where task synchronization is required, and scheduling concurrent tasks in Ada.

11.4.2 Scheduling Periodic Tasks

Initially, algorithms were developed for independent periodic tasks, which do not communicate or synchronize with each other [Liu73]. Since then, the theory has been developed considerably so that it can now be applied to practical problems, as will be illustrated in the examples. In this chapter, it is necessary to start with the basic rate monotonic theory for independent periodic tasks in order to understand how it has been extended to address more complex situations.

A periodic task has a period T, which is the frequency with which it executes, and an execution time C, which is the CPU time required during the period. Its CPU utilization U is the ratio C/T. A task is schedulable if all its deadlines are met (i.e., the task completes its execution before its period elapses). A group of tasks is considered to be schedulable if each task can meet its deadlines.

For a set of independent periodic tasks, the rate monotonic algorithm assigns each task a fixed priority based on its period, such that the shorter the period of a task, the higher its priority. Thus, if there are three tasks t_a, t_b, and t_c with periods 10, 20, 30 msec respectively, then the highest priority is given to the task with the shortest period

To do this, it is necessary to check the end of the first period of a given task t_i, as well as the end of all periods of higher priority tasks. Following the rate monotonic theory, these tasks will have shorter periods than t_i. These periods are referred to as scheduling points. Task t_i will execute once for a CPU amount of C_i during its period T_i. However, higher priority tasks will execute more often and can pre-empt t_i at least once. It is therefore necessary to consider the CPU time used up by the higher priority tasks as well.

Theorem 2 can be illustrated graphically using a timing diagram. Consider the example given earlier of the three tasks with the following characteristics:

Task t_1: $C_1 = 20$; $T_1 = 100$; $U_1 = 0.2$
Task t_2: $C_2 = 30$; $T_2 = 150$; $U_2 = 0.2$
Task t_3: $C_3 = 90$; $T_3 = 200$; $U_3 = 0.45$

The execution of the three tasks is illustrated using the timing diagram shown in Fig. 11.1.

Given the worst case of the three tasks being ready to execute at the same time, t_1 executes first because it has the shortest period and hence the highest priority. It completes after 20 msec after which the task t_2 executes for 30 msec. On completion of t_2, t_3 executes. At the end of the first scheduling point, $T_1 = 100$, which corresponds to t_1's deadline, t_1 has already completed execution and thus met its deadline. t_2 has also completed execution and easily met its deadline, and t_3 has executed for 50 msec out of the necessary 90.

At the start of t_1's second period, t_3 is pre-empted by task t_1. After executing for 20 msec, t_1 completes and relinquishes the CPU to task t_3 again. Then t_3 executes until the end of period T_2 (150 msec), which represents the second scheduling point due to t_2's deadline. As t_2 was completed before T_1 (which is less than T_2) elapsed, it easily met its deadline. At this time, t_3 has used up 80 msec out of the necessary 90.

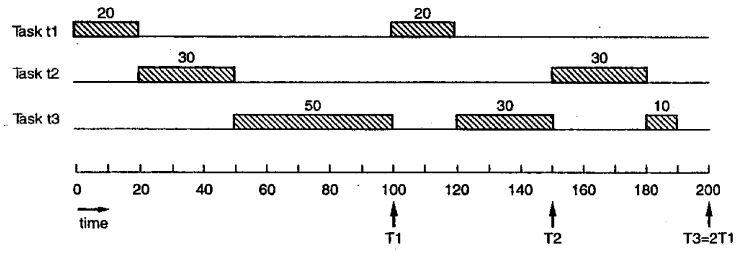

FIGURE 11.1 Timing Diagram

Task t_3 is pre-empted by task t_2 at the start of t_2's second period. After executing for 30 msec, t_2 completes relinquishing the CPU to task t_3 again. Task t_3 executes for another 10 msec, at which time it has used up all its CPU time of 90 msec, thereby completing before its deadline. Figure 11.1 shows the third scheduling point, which is both the end of t_1's second period ($2T_1 = 200$) and the end of t_3's first period ($T_3 = 200$). It also shows that each of the three tasks completed execution before the end of its first period and thus successfully met its deadline.

Figure 11.1 shows that the CPU is idle for 10 msec before the start of t_1's third period (also the start of t_3's second period). It should be noted that a total CPU time of 190 msec was used up over the 200 msec period, giving a CPU utilization for the above 200 msec of 0.95, although the overall utilization is 0.85. After an elapsed time equal to the least common multiple of the three periods (600 msec in this example) the utilization averages out to 0.85.

11.4.5 Mathematical Formulation of Completion Time Theorem

The Completion Time Theorem can be expressed mathematically in Theorem 3 [Sha90] as follows:

THEOREM 3

A set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if and only if:

$$\forall i,\ 1 \le i \le n,\quad \min_{(k,p) \in R_i} \sum_{j=1}^{i} C_j \frac{1}{pT_k} \left\lceil \frac{pT_k}{T_j} \right\rceil \le 1$$

where C_j and T_j are the execution time and period of task t_j respectively and $R_i = \{(k, p) \mid 1 \le k \le i, p = 1, \cdots, \lfloor T_i/T_k \rfloor\}$.

In the formula, t_i denotes the task to be checked and t_k denotes each of the higher priority tasks that impact the completion time of task t_i. For a given task t_i and a given task t_k, each value of p represents the scheduling points of task t_k. At each scheduling point, it is necessary to consider task t_i's CPU time C_i once, as well as the CPU time used by the higher priority tasks, and hence to determine whether t_i can complete its execution by that scheduling point.

Consider Theorem 3 applied to the three tasks, which were illustrated using the timing diagram in Fig. 11.1. The timing diagram is a graphical representation of what Theorem 3 computes. Once again the worst case is considered of the three tasks being

A second case often happens when there are aperiodic tasks. As discussed in Section 11.4.6, aperiodic tasks can be treated as periodic tasks with the worst case interarrival time considered the equivalent periodic task's period. Following the rate monotonic scheduling algorithm, if the aperiodic task has a longer period than a periodic task, it should execute at a lower priority than the periodic task. However, if the aperiodic task is interrupt driven, it will need to execute as soon as the interrupt arrives, even if its worst case interarrival time, and hence equivalent period, is longer than that of the periodic task.

The term priority inversion is given to any case where a task cannot execute because it is blocked by a lower priority task. In the case of rate monotonic priority inversion, the term "priority" refers to the **rate monotonic priority;** that is, the priority assigned a task based entirely on the length of its period and not on its relative importance. A task may be assigned an actual priority that is different from the rate monotonic priority. In this situation, **rate monotonic priority inversion** is used to refer to a task A pre-empted by a higher priority task B, where in fact task B's rate monotonic priority is lower than A's.

This is illustrated by the following example of rate monotonic priority inversion, in which there is a periodic task with a period of 25 msec and an interrupt driven task with a worst case interrarrival time of 50 msec. The periodic task has the higher rate monotonic priority since it has the shorter period. However, in practice, it is preferable to give the interrupt driven task the higher actual priority so that it can service the interrupt as soon as it arrives. Whenever the interrupt-driven task pre-empts the periodic task, this is considered a case of priority inversion, relative to the rate monotonic priority assignment, since if the interrupt driven task had been given its rate monotonic priority, it would not have pre-empted the periodic task.

It is necessary to extend the basic rate monotonic scheduling theory in order to address these practical cases of rate monotonic priority inversion. This has been achieved [SEI93] by extending the basic algorithms to take into account the blocking effect from lower priority tasks as well as pre-emption by higher priority tasks which do not observe rate monotonic priorities. As rate monotonic scheduling theory assumes rate monotonic priorities, pre-emption by higher priority tasks that do not observe the rate monotonic priorities is treated in a similar way to blocking by lower priority tasks.

Consider a task t_i with a period T_i during which it consumes C_i units of CPU time. The extensions to Theorems 1–3 mean that it is necessary to explicitly check each task t_i to determine whether it can meet its first deadline by considering:

a. Pre-emption time by higher priority tasks with periods less than t_i. These tasks can pre-empt t_i many times. Call this set H_n and let there be j tasks in this set. Let C_j be the CPU time for task t_j and T_j the period of task t_j, where $T_j < T_i$, the period of task t_i. The utilization of a task t_j in the H_n set is given by C_j/T_j.

b. Execution time for the task t_i. Task t_i executes once during its period T_i and consumes C_i units of CPU time.

c. Pre-emption by higher priority tasks with longer periods. These are tasks with nonrate monotonic priorities. They can only pre-empt t_i once as they have longer periods than t_i. Call this set H_1 and let there be k tasks in this set. Let the CPU time used by a task in this set be C_k. The worst case utilization of a task t_k in the H_1 set is given by C_k/T_i, since this means that t_k pre-empts t_i and uses up all its CPU time C_k during the period T_i.

d. Blocking time by lower priority tasks, as described in the previous section. These tasks can also only execute once as they have longer periods. Blocking delays have to be analyzed on an individual basis for each task ti to determine the worst case blocking situation for it. If B_i is the worst case blocking time for a given task t_i, then the blocking utilization for the period T_i is B_i/T_i.

Since for any given task t_i, cases a and b are taken care of by Theorems 1–3, the generalization of Theorems 1–3 is to take into account cases c and d. Theorem 1, the Utilization Bound Theorem, is extended to address all four cases above as follows:

$$U_i = \left(\sum_{j \in H_n} \frac{C_j}{T_j} \right) + \frac{1}{T_i} \left(C_i + B_i + \sum_{k \in H_1} C_k \right)$$

The Generalized Utilization Bound Theorem is referred to here as Theorem 4. U_i is the utilization bound during a period T_i for task t_i. The first term in Theorem 4 is the total pre-emption utilization by higher priority tasks with periods less than t_i. The second term is the CPU utilization by task t_i. The third term is the worst case blocking utilization experienced by t_i. The fourth term is the total pre-emption utilization by higher priority tasks with longer periods than t_i.

By substituting in the above equation the utilization U_i can be determined for a given task. If U_i is less than the worst case upper bound (Table 11.1), this means that the task t_i will meet its deadline. It is important to realize that the utilization bound test needs to be applied to each task, since in this generalized theory, where rate monotonic priorities are not necessarily observed, the fact that a given task meets its deadline is no guarantee that a higher priority task will meet its deadline.

Once again if the utilization bound test fails, a more precise test is available, which verifies whether or not each task can complete execution during its period. This is a generalization of the Completion Time Theorem, and assuming all tasks are ready for execution at the start of a task t_i's period, determines whether t_i can complete execution by the end of its period, given pre-emption by higher priority tasks and blocking by lower priority tasks. Pictorially, this can be illustrated by drawing a timing diagram for all the tasks up to the end of task t_i's period T_i.

11.4.9 Real-time Scheduling and Design

Real-time scheduling theory can be applied to a set of concurrent tasks at the design stage or after the tasks have been implemented. In this book, the emphasis is on

b. Execution time C_3 for the task t_3. Execution utilization = $U_3 = 0.1$.

c. Pre-emption by higher priority tasks with longer periods. There are no tasks that fall into this category.

d. Blocking time by lower priority tasks. There are no tasks that fall into this category.

Worst case utilization = Pre-emption utilization + Execution utilization = 0.32 + 0.1 = 0.42 < worst case upper bound of 0.69. Consequently t_3 will meet its deadline. In conclusion, all four tasks will meet their deadlines.

11.4.11 Real-time Scheduling in Ada

There are some aspects of the Ada conceptual model that raise concerns about its suitability for real-time systems that support tasks with hard deadlines. For example, tasks are queued FIFO rather than by priority, task priorities cannot be dynamically changed at run-time, and priority inversion can lead to high priority tasks being delayed indefinitely by lower priority tasks.

These problems are being addressed by proposing changes to the Ada run-time system to support the priority ceiling protocol and by providing a comprehensive set of guidelines for Ada real-time programming together with an "enlightened interpretation of Ada's scheduling rules [Sha90]."

11.5 Performance Analysis Using Event Sequence Analysis

During the requirements phase of the project, the system's required response times to external events are specified. After task structuring, a first attempt at allocating time budgets to the concurrent tasks in the system can be made. Event sequence analysis is used to determine the tasks that need to be executed to service a given external event. An event sequence diagram is used to show the sequence of internal events and tasks activated following the arrival of the external event. The approach is described next.

Consider an external event. Determine which I/O task is activated by this event, and then determine the sequence of internal events that follow. This necessitates identifying the tasks that are activated and those I/O tasks that generate the system response to the external event. Estimate the CPU time for each task. Estimate the CPU overhead, which consists of context switching overhead, interrupt handling overhead, and intertask communication and synchronization overhead. It is also necessary to consider any other tasks that execute during this period. The sum of the CPU times for the tasks that participate in the event sequence, plus any additional tasks that execute, plus CPU overhead, must be less or equal to the specified system response

time. If there is some uncertainty over the CPU time for each task, then allocate a worst case upper bound.

To estimate overall CPU utilization, it is necessary to estimate, for a given time interval, the CPU time for each task. If there is more than one path through the task, then estimate the CPU time for each path. Next, estimate the frequency of activation of tasks. This is easily computed for periodic tasks. For asynchronous tasks, consider the average and maximum activation rates. Multiply each task's CPU time by its activation rate. Sum all the task CPU times, and then compute CPU utilization. (An example of applying the event sequence analysis approach is given in Chapter 19.)

11.6 Performance Analysis Using Real-time Scheduling Theory and Event Sequence Analysis

This section describes how the real-time scheduling theory can be combined with the event sequence analysis approach. Instead of considering individual tasks, it is necessary to consider all the tasks in an event sequence. The task activated by the external event executes first and then initiates a series of internal events, resulting in other internal tasks being activated and executed. It is necessary to determine whether all the tasks in the event sequence can be executed before the deadline.

Initially attempt to allocate all the tasks in the event sequence the same priority. These tasks can then collectively be considered one equivalent task from a real-time scheduling viewpoint. This equivalent task has a CPU time equal to the sum of the CPU times of the tasks in the event sequence, plus context switching overhead, plus message communication or event synchronization overhead. The worst case interarrival time of the external event that initiates the event sequence is then made the period of this equivalent task.

To determine whether the equivalent task can meet its deadline, it is necessary to apply the real-time scheduling theorems. In particular it is necessary to consider pre-emption by higher priority tasks, blocking by lower priority tasks, as well as the execution time of this equivalent task. An example of combining event sequence analysis with real-time scheduling using the equivalent task approach is given in Chapter 19 for the cruise control problem.

In some cases, the approach of assuming that all the tasks in the event sequence can be replaced by an equivalent task cannot be used (e.g., one of the tasks is used in more than one event sequence, or executing the equivalent task at that priority would prevent other tasks from meeting their deadlines). In that case, the tasks in the event sequence need to be analyzed separately and assigned different priorities. In determining whether the tasks in the event sequence will meet their deadline, it is necessary to consider pre-emption and blocking on a per task basis. However, it is still necessary to determine whether all tasks in the event sequence will complete before the deadline.