1.2 Time, Performance, and Quality of Service

In 2002, the Object Management Group (OMG) adopted the Unified Modeling Language™ (UML) profile that provided a standardized means for specifying timeliness, performance, and schedulability aspects of systems and parts of systems, the so-called Real-Time Profile (RTP) [7]. In the same year, an initial submission was made for a UML profile that detailed a generalized framework for Quality of Service (QoS) aspects of which timeliness, performance, and schedulability are common examples as well as a specific profile for fault tolerance mechanisms [8]. To be sure, these profiles are clearly not necessary for modeling the QoS aspects developers have been doing this for as long as UML has been around (OK, even longer!); however, codifying what the best developers were already doing and defining specific names for the aspects, allows models to be easily exchanged among different kinds of tools, such as UML design and schedulability analysis tools.

These profiles provide a minutely specialized form of UML dealing with the issues important to some domain of concern. In this case, of course, the domain of concern is real-time, high-reliability, and safety-critical systems. These specializations take the form of representation of domain concepts as stereotypes of standard UML elements, with additional tag values for the QoS aspects, whose values can be specified in constraints. The details of the profiles will be discussed later. For now, let's limit our concern to the domain concepts themselves and not to how they are presented in the profiles.

1.2.1 Modeling Actions and Concurrency

Two critical aspects of real-time systems are how time is handled and the execution of actions with respect to time. The design of a real-time system must identify the timing requirements of the system and ensure that the system performance is both logically correct and timely.

Most developers concerned with timeliness of behavior express it in terms of

actual execution time relative to a fixed budget called a time constraint. Often, the constraint is specified as a deadline, a single time value (specified from the onset of an initiating stimulus) by which the resulting action must complete. The two types of time constraints commonly used are hard and soft:

- **Hard** — The correctness of an action includes a description of timeliness. A late completion of an action is incorrect and constitutes a system failure. For example, a cardiac pacemaker must pace outside specific periods of time following a contraction or fibrillation.[8]

- **Soft** — For the most part, software requirements are placed on a set of instance executions rather than on each specific execution. Average deadline and average throughput are common terms in soft real-time systems. Sometimes, missing an entire action execution may be acceptable. Soft real-time requirements are most often specified in probabilistic or stochastic terms so that they apply to a population of executions rather than to a single instance of an action execution. It is common to specify but not validate soft real-time requirements. An example of a soft real-time specification might be that the system must process 1200 events per second on average, or that the average processing time of an incoming event is 0.8 ms.

[8] Uncoordinated contraction of random myocardial cells. This is a bad thing.

Usually a timing constraint is specified as a deadline, a milestone in time that somehow constrains the action execution.

In the RTP, actions may have the time-related properties shown in Table 1-1.

As Table 1-1 suggests, the basic concepts of timeliness in real-time systems are straightforward even if their analysis is not. Most time requirements come from bounds on the performance of reactive systems. The system must react in a timely way to external events. The reaction may be a simple digital actuation, such as turning on a light, or a complicated loop controlling dozens of actuators simultaneously. Typically, many subroutines or tasks must execute between the causative event and the resulting system action. External requirements bound the overall performance of the control path. Each of the processing activities in the control path is assigned a portion of the overall time budget. The sum[9] of the time budgets for any path must be less than or equal to the overall performance constraint.

[9] Although due to blocking and preemption, it is not generally just a simple arithmetic sum.

Because of the ease of analysis of hard deadlines, most timeliness analysis is done assuming hard deadlines and worst-case completion times. However, this can lead to overdesign of the hardware at a potentially much greater recurring cost than if a more detailed analysis were done.

Table 1-1. Timeliness Properties of Actions

Attribute	Description
Priority	The priority of the action from a scheduling perspective. It may be set as a result of static analysis or by dynamic scheduling software.
Blocking Time	The length of time that the action is blocked waiting for resources.
Ready Time	The effective Release Time expressed as the length of time since the beginning of a period; in effect a delay between the time an entity is eligible for execution and the actual beginning of execution.
Delay Time	The length of time an action that is eligible for execution waits while acquiring and releasing resources.
Release Time	The instant of time at which a scheduling job becomes eligible for execution.
Preempted Time	The length of time the action is preempted, when runable, to make way for a higher-priority action.
Worst-Case Completion Time	The overall time taken to execute the action, including overheads.
Laxity	Specifies the type of deadline, hard or soft.
Absolute Deadline	Specifies the final instant by which the action must be complete. This may be either a hard or a soft deadline.

Attribute	Description
Relative Deadline	For soft deadlines, specifies the desired time by which the action should be complete.
start	The start time of the action.
end	The completion time of the action.
duration	The total duration of the action (not used if start and end times are defined).
isAtomic	Identifies whether the action can be pre-empted or not.

In the design of real-time systems several time-related concepts must be identified and tracked. An action may be initiated by an external event. Real-time actions identify a concept of timely usually in terms of a deadline specified in terms of a completion time following the initiating event. An action that completes prior to that deadline is said to be timely; one completing after that deadline is said to be late.

Actions are ultimately initiated by events associated with the reception of messages arising from objects outside the scope of the system. These messages have various arrival patterns that govern how the various instances of the messages arrive over time. The two most common classifications of arrival patterns are periodic and aperiodic (i.e., episodic). Periodic messages may vary from their defined pattern in a (small) random way. This is known as jitter.

Aperiodic arrivals are further classified into bounded, bursty, irregular, and stochastic (unbounded). In bounded arrival times a subsequent arrival is bounded between a minimum interarrival time and a maximum interarrival time. A bursty message arrival pattern indicates that the messages tend to clump together in time (statistically speaking, there is a positive correlation in time between the arrival of one message and the near arrival of the next).[10] Bursty arrival patterns are characterized by a maximum burst length occurring within a specified burst interval. For example, a bouncy button might be characterized as giving up to 10 events within a 20ms burst interval. The maximum burst length is also sometimes specified with a probability density function (PDF) giving a likely distribution or frequency within the burst interval. Lastly, if the events arrivals are truly uncorrelated with each other, they may be

modeled as arising randomly from a probability density function (PDF). This is the stochastic arrival pattern.

> [10] Bursty message arrival patterns are characterized by a Poisson distribution and so do not have a standard deviation but do have an average interarrival time.

Knowing the arrival pattern of the messages leading to the execution of system actions is not enough to calculate schedulability, however. It is also important to know how long the action takes to execute. Here, too, a number of different measures are used in practice. It is very common to use a worst-case execution time in the analysis. This approach allows us to make very strong statements about absolute schedulability, but has disadvantages for the analysis of systems in which occasional lateness is either rare or tolerable. In the analysis of so-called soft real-time systems, it is more common to use average execution time to determine a statistic called mean lateness.

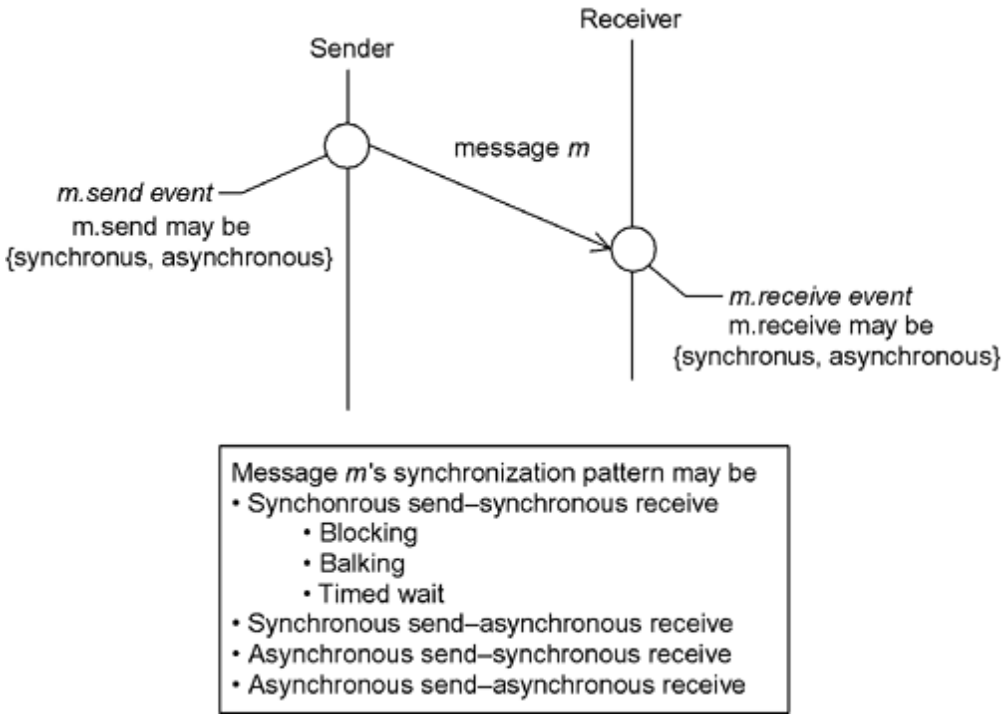
Actions often execute at the same time as other actions. We call this concurrency. Due to the complexity of systems, objects executing actions must send messages to communicate with objects executing other actions. In object terminology, a message is an abstraction of the communication between two objects. This may be realized in a variety of different ways. Synchronization patterns describe how different concurrent actions rendezvous and exchange messages. It is common to identify means for synchronizing an action initiated by a message sent between objects.

In modern real-time systems it cannot be assumed that the sender of a message and the receiver of that message are located in the same address space. They may be executing on the same processor, in the same thread, on the same processor but different threads, or on separate processors. This has implications for the ways in which the sender and receiver can exchange messages. The transmission of a message m may be thought of as having two events of interest a send event followed by a receive event. Each of these events may be processed in either a synchronous or an asynchronous fashion, leading to four fundamental synchronization patterns, as shown in Figure 1-1. The sending of an event is synchronous if the sender waits until the message is sent before continuing and asynchronous if not.[11] The reception of an event is synchronous if the receiver immediately processes it and asynchronous if that processing is delayed. An example of synch-synch pattern (synchronous send synchronous receive) is a standard function or

operation call. A message sent and received through a TCP/IP protocol stack is an example of an asynch-asynch transfer, because the message is typically queued to send and then queued during reception as well. A remote procedure call (RPC) is an example of a synch-asynch pattern; the sender waits until the receiver is ready and processes the message (and returns a result), but the receiver may not be ready when the message is sent.

[11] Note that some people consider an invocation synchronous if the caller waits until the receiver returns a response. This is what we would call the blocking kind of synch-synch rendezvous.

Figure 1-1. Synchronization Patterns



In addition to these basic types, a balking rendezvous is a synch-synch rendezvous that aborts the message transfer if the receiver is not immediately ready to receive it, while a timed wait rendezvous is a balking rendezvous in which the sender waits for a specified duration for the receiver to receive and process the message before aborting. A blocking rendezvous is a synch-synch rendezvous in which the sender will wait (forever if need be) until the receiver accepts and processes the message.

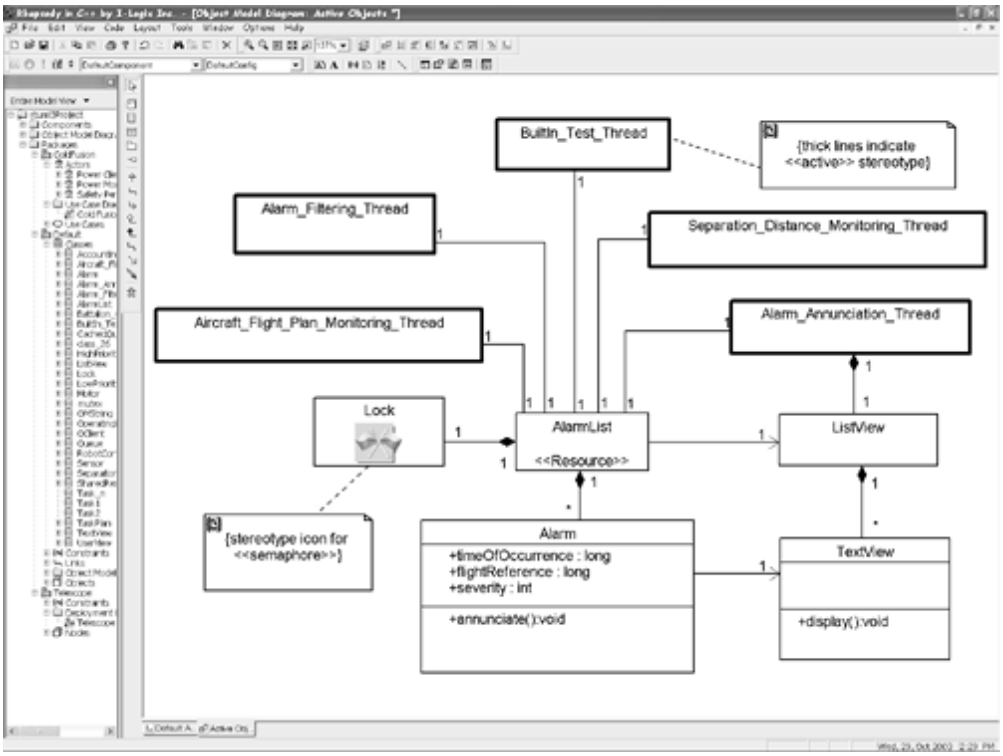
In lay terms, concurrency is doing more than one thing at once. When concurrent actions are independent, life is easy. Life is, however, hard when the actions must share information or rendezvous with other concurrent actions. This is where concurrency modeling becomes complex.

Programmers and models typically think of the units of concurrency as threads, tasks, or processes. These are OS concepts, and as such available as primitive services in the operating system. It is interesting that the RTP does not discuss these concepts directly. Instead, the RTP talks about an active resource (a resource that is capable of generating its own stimuli asynchronously from other activities). A concurrent Unit is a kind of active resource that associates with a scenario (a sequenced set of actions that it executes) and owns at least one queue for holding incoming stimuli that are waiting to be accepted and processed. Sounds like a thread to me ;-)

As a practical matter, «active» objects contain such concurrentUnits, have the responsibility to start and stop their execution, and, just as important, delegate incoming stimuli to objects contained within the «active» objects via the composition relationship. What this means is that we will create an «active» object for each thread we want to run, and we will add passive (nonactive) objects to it via composition when we want them to execute in the context of that thread. Figure 1-2 shows a typical task diagram in UML.[12] The boxes with thick borders are the «active» objects. The resource is shown with the annotation «resource», called a stereotype. In the figure, an icon is used to indicate the semaphore rather than the alternative «stereotype» annotation. We will discuss the UML notions of objects and relations in much more detail in the next chapter.

[12] A task diagram is just a class diagram (see Chapter 2) whose mission is to show the classes related to the concurrency of the system.

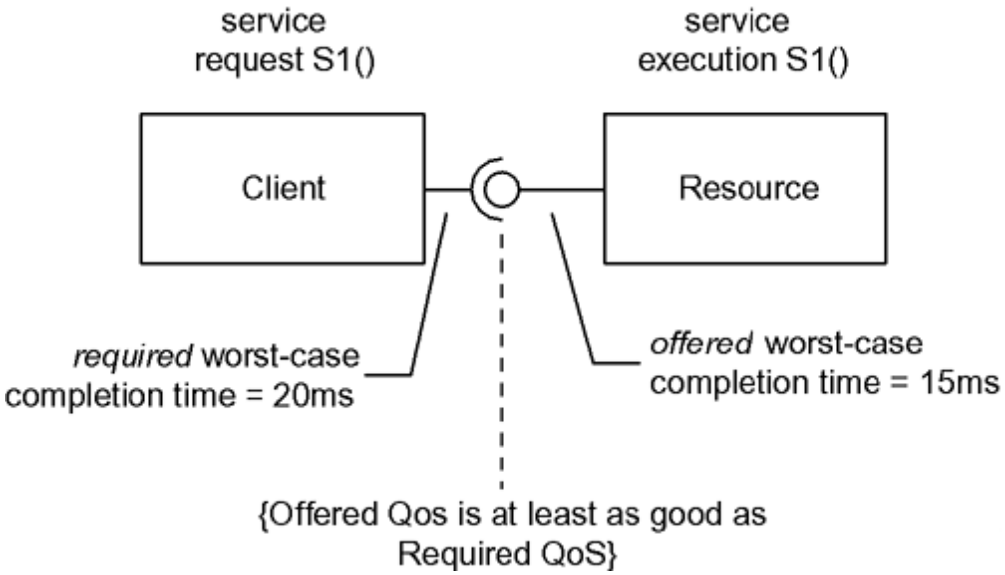
Figure 1-2. «active» Objects and Threads



1.2.2 Modeling Resources

A simple analysis of execution times is usually inadequate to determine schedulability (the ability of an action to always meet its timing constraints). Most timeliness analysis is based, explicitly or implicitly, on a client-resource model in which a client requires a specified QoS from a resource and a resource offers a QoS. Assuming the offered and required values are correct, schedulability analysis is primarily a job of ensuring that offered QoS is always at least as good as required. See Figure 1-3.

Figure 1-3. Required and Offered Quality of Service



A resource from [7] is defined to be "an element whose service capacity is limited, directly or indirectly, by the finite capacities of the underlying physical computing environment." Resource is a fundamental concept, one that appears many times in real-time systems. Much of the design of a real-time system is devoted to ensuring that resources act in accordance with their preconditional invariant assumptions about exclusivity of access and quality of service.

The execution times used in the analysis must take into account the blocking time, that is, the length of time an action of a specific priority is blocked from completion by a lower-priority action owning a required resource.

1.2.3 Modeling Time

At the best of, uh, times, time itself is an elusive concept. While we lay folks talk about it passing, with concepts like past, present, and future, most philosophers reject the notion. There seem to be two schools of thought on the issue. The first holds that time is a landscape with a fixed past and future. Others feel it is a relationship between system (world) states as causal rules play out. Whatever it may be,[13] with respect to real-time systems, we are concerned with capturing the passage of time using milestones (usually deadlines) and measuring time. So we will treat time as if it flowed and we could measure it.

> [13] And I, for one, feel like St. Augustine of Hippo who said he knew what time was until someone asked him to explain it.

The RTP treats time as an ordered series of time instants. A timeValue

corresponds to such a physical instant of time and may be either dense (meaning that a timeValue exists between any two timeValues) or discrete (meaning there exist timeValue pairs that do not have another timeValue between them). A timeValue measures one or more physical instants. A duration is the interval between two physical instants; that is, it has both starting and ending physical instants. A timeInterval may be either absolute (same as a duration) or relative. So what's the difference between a duration and a timeInterval? A duration has a start and end physical instant, while a timeInterval has a start and end timeValue. Confusing? You bet!

Fortunately, the concepts are not as difficult to use in practice as they are to define precisely. In models, you'll always refer to timeValues because that's what you measure, so timeIntervals will be used more than durations.

Speaking of measuring time....We use two mechanisms to measure time: clocks and timers. All time is with reference to some clock that has a starting origin. It might be the standard calendar clock, or it might be time since reboot. If there are multiple clocks, they will be slightly different, so in distributed systems we have to worry about synchronizing them. Clocks can differ in a couple of ways. They can have a different reference point and be out of synch for that reason this is called clock offset. They may also progress at different rates, which is known as skew. Skew, of course can change over time, and that is known as clock drift. Offset, skew, and drift of a timing mechanism are all with respect to a reference clock. Clocks can be read, set, or reset (to an initial state), but mostly they must march merrily along. Timers, on the other hand, have an origin (as do clocks) but they also generate timeout events. The current time of a timer is the amount of time that must elapse before a timeout occurs. A timer is always associated with a particular clock. A retriggerable timer resets after a timeout event and starts again from its nominal timeout value, while a nonretriggerable timer generates a single timeout event and then stops.

There are two kinds of time events: a timeout event and a clock interrupt. A clock interrupt is a periodic event generated by the clock representing the fundamental timing frequency. A timeout event is the result of achieving a specified time from the start of the timer.

1.2.4 Modeling Schedulability

It might seem odd at first, but the concepts of concurrency are not the same as the concepts of scheduling. Scheduling is a more detailed view with the

responsibility of executing the mechanisms necessary to make concurrency happen. A scheduling context includes an execution engine (scheduler) that executes a scheduling policy, such as round robin, cyclic executive, or preemptive. In addition, the execution engine owns a number of resources, some of which are scheduled in accordance with the policy in force. These resources have actions, and the actions execute at some priority with respect to other actions in the same or other resources.

Operating systems, of course, provide the execution engine and the policies from which we select the one(s) we want to execute. The developer provides the schedulable resources in the forms of tasks («active» objects) and resources that may or may not be protected from simultaneous access via mechanisms such as monitors or semaphores. Such systems are fairly easy to put together, but there is (or should be!) concern about the schedulability of the system, that is, whether or not the system can be guaranteed to meet its timeliness requirements.

Determining a scheduling strategy is crucial for efficient scheduling of real-time systems. Systems no more loaded than 30% have failed because of poorly chosen scheduling policies.[14] Scheduling policies may be stable, optimal, responsive, and/or robust. A stable policy is one in which, in an overload situation, it is possible to a priori predict which task(s) will miss their timeliness requirements. Policies may also be optimal.[15] A responsive policy is one in which incoming events are handled in a timely way. Lastly, by robust, we mean that the timeliness of one task is not affected by the misbehavior of another. For example, in a round robin scheduling policy, a single misbehaving task can prevent any other task in the system from running.

[14] Doug Locke, Chief Scientist for TimeSys, private communication.

[15] An optimal policy is one that can schedule a task set if it is possible for any other policy to do so.

Many different kinds of scheduling policies are used. Scheduling policies can be divided into two categories: fair policies and priority policies. The first category schedules things in such a way that all tasks progress more or less evenly. Examples of fair policies are shown in Table 1-2.

Table 1-2. Fair Scheduling Policies

Scheduling Policy	Description	Pros	Cons
Cyclic Executive	The scheduler runs a set of tasks (each to completion) in a never-ending cycle. Task set is fixed at startup.	∘ Fair ∘ Very simple ∘ Highly predictable	∘ Unresponsive ∘ Unstable ∘ Nonoptimal ∘ Nonrobust ∘ Requires tuning ∘ Short tasks[16]
Time-Triggered Cyclic Executive	Same as cyclic executive except that the start of a cyclic is begun in response to a time event so that the system pauses between cycles.	∘ Fair ∘ Very simple ∘ Highly predictable ∘ Resynchronizes cycle with reference clock	∘ Unresponsive ∘ Unstable ∘ Nonoptimal ∘ Nonrobust ∘ Requires tuning ∘ Short tasks

Scheduling Policy	Description	Pros	Cons
Round Robin	A task, once started, runs until it voluntarily relinquishes control to the scheduler. Tasks may be spawned or killed during the run.	o Fair o More flexible than cyclic executive o Simple	o Unresponsive o Unstable o Nonoptimal o Nonrobust o Short tasks
Time-Division Round Robin	A round robin in which each task, if it does not relinquish control voluntarily, is interrupted within a specified time period, called a time slice.	o Fair o More flexible than cyclic executive or round robin o Simple o Robust	o Unresponsive o Unstable o Nonoptimal

[16] By short tasks we mean that for the policy to be fair, each task must execute for a relatively short period of time. Often a task that takes a long time to run must be divided by the developer into shorter blocks to achieve fairness. This places an additional burden on the developer.

In contrast, priority-driven policies are unfair because some tasks (those of higher priority) are scheduled preferentially to others. In a priority schedule, the priority is used to select which task will run when more than one task is ready to run. In a preemptive priority schedule, when a ready task has a

priority higher than that of the running task, the scheduler preempts the running task (and places it in a queue of tasks ready to run, commonly known as the "ready queue") and runs the highest-priority ready task. Priority schedulers are responsive to incoming events as long as the priority of the task triggered by the event is a higher priority than the currently running tasks. In such systems, interrupts are usually given the highest priority.

Two competing concepts are importance and urgency. Importance refers to the value of a specific action's completion to correct system performance. Certainly, correctly adjusting the control surface of an aircraft in a timely manner is of greater importance than providing flicker-free video on a cabin DVD display. The urgency of an action refers to the nearness of its deadline for that action without regard to its importance. The urgency of displaying the next video frame might be much higher than the moment-by-moment actuation control of the control surfaces, for example. It is possible to have highly important, yet not urgent actions, and highly urgent but not important ones mixed freely within a single system. Most scheduling executives, however, provide only a single means for scheduling actions priority. Priority is an implementation-level solution offered to manage both importance and urgency.

All of the priority-based scheduling schemes in Table 1-3 are based on urgency. It is possible to also weight them by importance by multiplying them by an importance factor w_j and then set the task priority equal to the product of the unweighted priority and the importance factor. For example, in RMS scheduling the task priority would be set to

Table 1-3. Priority Scheduling Policies

Scheduling Policy	Description	Pros	Cons
Rate Monotonic Scheduling (RMS)	All tasks are assumed periodic, with their deadline at the end of the period. Priorities are assigned as design time so that tasks with the shortest periods have the highest priority.	○ Stable ○ Optimal ○ Robust	○ Unfair ○ May not scale up to highly complex systems
Deadline Monotonic	Same as RMS except it is not assumed that the	○ Stable	○ Unfair

Scheduling Policy	Description	Pros	Cons
Scheduling (DMS)	deadline is at the end of the period, and priorities are assigned at design time, based on the shortness of the task's deadline.	○ Optimal ○ Robust	○ Handles tasks more flexibly
Maximum Urgency First (MUF)	Priorities are assigned at runtime when the task becomes ready to run, based on the nearness of the task deadlines the nearer the deadline, the higher the priority.	○ Optimal ○ Scales up better than RMS or DMS ○ Robust	○ Unstable ○ Unfair ○ Lack of RTOS support
Earliest Deadline Scheduling (EDS)	Laxity is defined to be the time-to-deadline minus the remaining task-execution-time. LL scheduling assigns higher priorities to lower laxity values.	○ Robust ○ Optimal	○ In naïve implementation, causes thrashing ○ Unstable ○ Unfair ○ Even less RTOS support than EDS ○ More complex
Least Laxity (LL)	MUF is a hybrid of LL and RMS. A critical task set is run using the highest set of priorities under an RMS	○ Robust ○	○ Critical task set runs preferentially to other tasks, to some stability

Scheduling Policy	Description	Pros	Cons
	schedule, and the remaining (less critical) tasks run at lower priorities, scheduled using LL.	Optimal	is achieved, although not for the LL task set o In naïve implementation, causes thrashing o Unstable o Unfair o Even less RTOS support than EDS o More complex

Equation 1-1 Importance-Weighted Priority

$$[DISEQ] p_j = w_j / C_j$$

Where p_j is the priority of task j, w_j is some measure of the importance of the completion of the task, and C_j is the period of task j. [10] provides a reasonably rigorous treatment of weighted scheduling policies for generalized scheduling algorithms. Note that some operating systems use ascending priority values to indicate higher priority while others use descending priorities to indicate higher priority.

Since essentially all real-time systems running multiple tasks must coordinate those tasks and share resources, the manner in which those resources are managed is crucial not only to the correct operation of the system but also to its schedulability. The fundamental issue is the protection of the resource from corruption due to interleaved writing of the values or from a reader getting an incorrect value because it read partway through an update from another task.

Of course, for physical resources, there may be other integrity concerns as well, due to the nature of the physical process being managed. The most common basic solution is to serialize the access to the resource and prevent simultaneous access. There are a number of technical means to accomplish this. For example, access can be made atomic that is, no other task is allowed to run when one task owns a resource. Such accesses are made during what is called a critical section. This approach completely prevents simultaneous access but is best suited when the access times for the resource are very short relative to the action completion and deadline times [11]. Atomic access also breaks the priority-based scheduling assumption of infinite preemptibility (that is, when a task will run as soon as it is ready to run and the highest priority ready task in the system). However, if the critical section isn't too long, then a mild violation of the assumption won't affect schedulability appreciably. When a resource must be used for longer periods of time, the use of critical sections is not recommended.

Another common approach to access serialization is to queue the requests. That is useful when the running task doesn't have to precisely synchronize with the resource, but wants to send it a message. Queuing a message is a "send-and-forget" kind of rendezvous that works well for some problems and not for others, depending on whether the resource (typically running in its own thread in this case) and the client must be tightly coupled with respect to time. For example, sending a time-stamped log entry to a logging database via a queue allows the sending task to send the message and go on about its business, knowing that eventually the database will handle the log entry. This would not be appropriate in a real-time control algorithm in which a sensor value is being used to control a medical laser, however.

A third common approach to access serialization is protect the resource with a mutual exclusion (mutex) semaphore. The semaphore is an OS object that protects a resource. When access is made to an unlocked resource, the semaphore allows the access and locks it. Should another task come along and attempt to access the resource, the semaphore detects it and blocks the second task from accessing the resource the OS suspends the task allowing the original resource client to complete. Once complete, the resource is unlocked, and the OS then allows the highest-priority task waiting on that resource to access it, while preventing other tasks from accessing that resource. When a task is allowed to run (because it owns a needed resource) even though a higher-priority task is ready to run, the higher-priority task is said to be blocked.

While this approach works very well, it presents a problem for schedulability called unbounded priority inversion. When a task is blocked, the system is said to be in a condition of priority inversion because a lower-priority task is running even though a higher-priority task is ready to run. In Figure 1-4, we see two tasks sharing a resource,[17] HighPriorityTask and the LowPriorityTask. In the model presented here, a low-priority value means the task is a high priority. Note that there is a set of tasks on intermediate priority, Task1 through Task_n, and that these tasks do not require the resource. Therefore, if LowPriorityTask runs and locks the resource, and then HighPriorityTask wants to run, it must block to allow LowPriorityTask to complete its use of the resource and release it. However, the other tasks in the system are of higher priority than LowPriorityTask and then can (and will) preempt LowPriority Task when they become ready to run. This, in effect, means that the highest priority task in the system is blocked not only by the LowPriority Task, but potentially by every other task in the system. Because there is no bound on the number of these intermediate priority tasks, this is called unbounded priority inversion. There are strategies for bounding the priority inversion and most of these are based on the temporary elevation of the priority of the blocking task (in this case, LowPriority Task). Figure 1-5 illustrates the problem in the simple implementation of semaphore-based blocking.

[17] The notations will be covered in the next chapter. For now, the boxes represent "things" such as tasks, resources, and semaphores, and the lines mean that connected things can exchange messages. The notes in curly braces, called constraints, provide additional information or annotations.

Figure 1-4. Priority Inversion Model

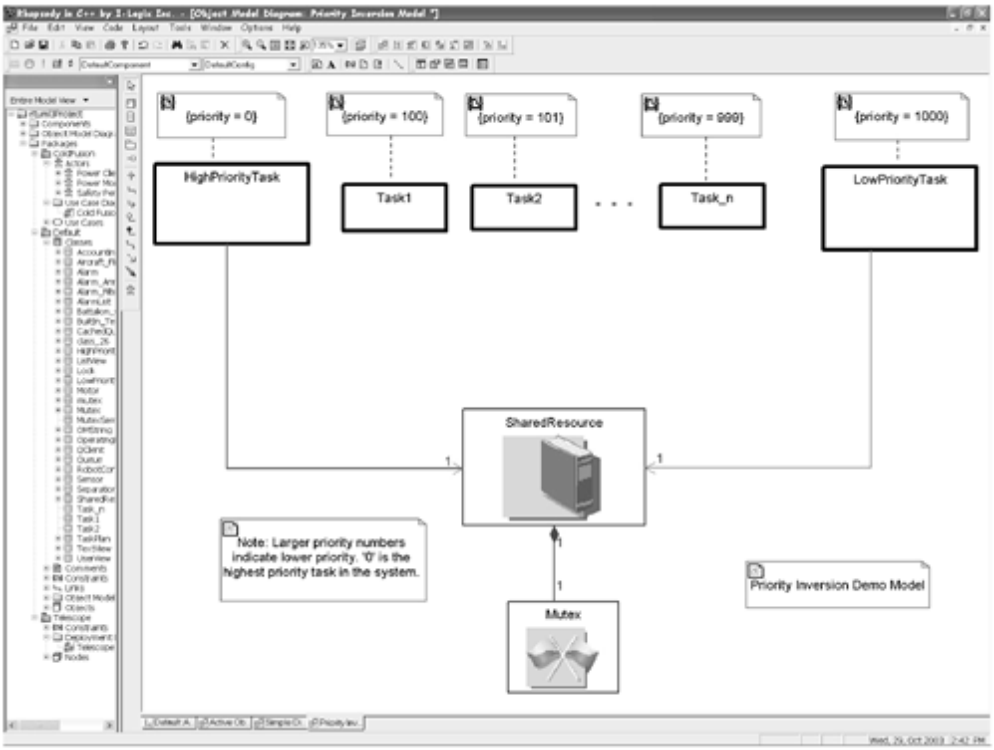
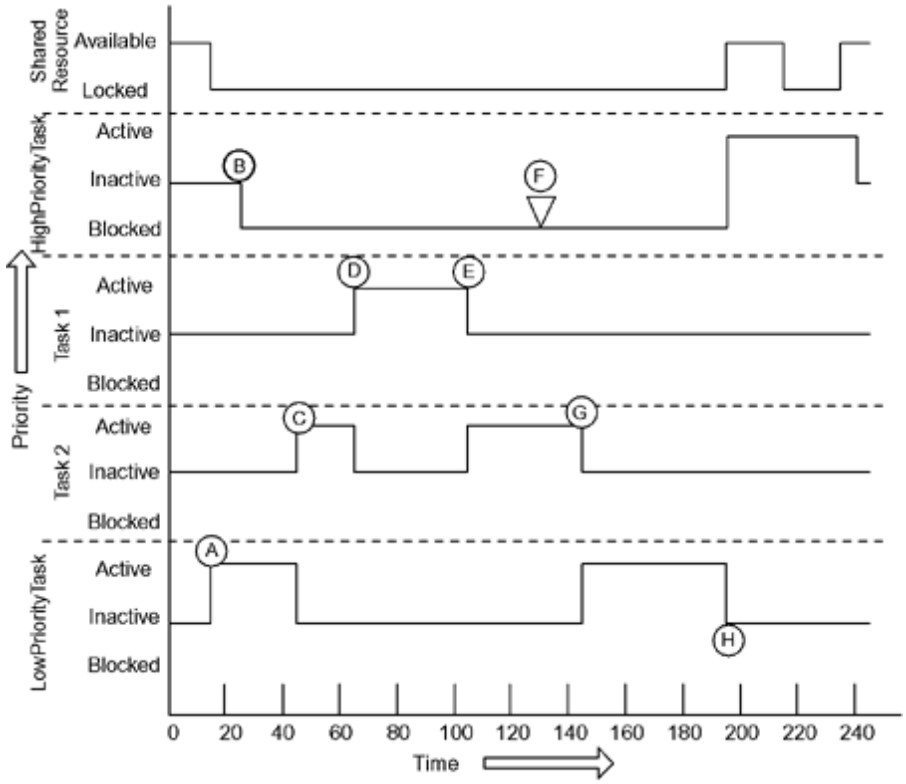


Figure 1-5. Priority Inversion Scenario



Legend:
Priorities: HighPriorityTask (HPT) > Task 1 > Task 2 > LowPriorityTask (LPT)
A: LPT is ready to run and starts and during execution, locks the shared resource (SR).
B: HPT is ready to run, but needs the resource. It is blocked and must allow Task 1 to complete.
C: Task 2, which is a higher priority than LPT, is ready to run. Since it doesn't need the resource, it preempts LPT. HPT is now effectively blocked by both LPT and Task 2.
D: Task 1, which is a higher priority than Task 2, is ready to run. Since it doesn't need the resource, it reempts Task 2. HPT is now effectively blocked by three tasks.
E: Task 1 completes, allowing Task 2 to resume.
F. HPT's deadline is missed even though it was long enough to allow the execution of both HPT and LPT. It was not long enough to allow all the other tasks to run, however.
G. Task 2 completes, allowing LPT to resume.
H. LPT (finally) completes and releases the resource, allowing Task 1 to access the resource.

As mentioned, there are a number of solutions to the unbounded priority inversion problem. In one of them, the Highest Locker Pattern of [11] (see also [9,12]), each resource has an additional attribute its priority ceiling. This is the priority just above that of the highest-priority task that can ever access the resource. The priority ceiling for each resource is determined at design time. When the resource is locked, the priority of the locking task is temporarily elevated to the priority ceiling of the resource. This prevents intermediate priority tasks from preempting the locking task as long as it owns the resource. When the resource is released, the locking task's priority is reduced to either its nominal priority or to the highest priority ceiling of any resources that remain locked by that task. There are other solutions with different pros and cons that the interested reader can review.

An important aspect of schedulability is determining whether or not a particular

task set can be scheduled, that is, can it be guaranteed to always meet its timeliness requirements? Because timeliness requirements are usually expressed as deadlines, that is the case we will consider here.

When a task set is scheduled using RMS, then certain assumptions are made. First, the tasks are time-driven (i.e., periodic). If the tasks are not periodic, it is common to model the aperiodic tasks using the minimum interarrival time as the period. While this works, it is often an overly strong condition that is, systems that fail this test may none theless be schedulable and always meet their deadlines, depending on the frequency with which the minimum arrival time actually occurs. The other assumptions include infinite preemptibility (meaning that a task will run immediately if it is now the highest-priority task ready to run) and the deadline occurs at the end of the period. In addition, the tasks are independent that is, there is no blocking. When these things are true, then Equation 1-2 provides a strong condition for schedulability. By strong, we mean that if the inequality is true, then the system is schedulable; however, just because it's not true does not imply necessarily that the system is not schedulable. More detailed analysis might be warranted for those cases.

Equation 1-2 Basic Rate Monotonic Analysis

$$\sum_{n} \frac{C_j}{T_j} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

In Equation 1-2, C_j is the execution time for the task. In a worst-case analysis, this must be the worst-case execution time (i.e., worst-case completion time). T_j is the period of the task, and n is the number of tasks. The ratio C_j/T_j is called the utilization of the task. The expression on the right side of the inequality is called the utilization bound for the task set (note that 2 is raised to the power of (1/n)). The utilization bound converges to about 0.69 as the number of tasks grows. It is less than 1.0 because in the worst case, the periods of the tasks are prime with respect to each other. If the task periods are all multiples of each other (for example, periods of 10, 100, 500), then a utilization bound of 100% can be used for this special case (also for the case of dynamic priority policies, such as EDS). As an example, consider the case in which we have four tasks, as described in Table 1-4.

Table 1-4. Sample Task Set

Task	Execution Time (ms)	Period (ms)	Cj/Tj

Task	Execution Time (ms)	Period (ms)	Cj/Tj
Task 1	10	100	0.1
Task 2	30	150	0.2
Task 3	50	250	0.2
Task 4	100	500	0.2

The sum of the utilizations from the table is 0.7. The utilization bound for four tasks is 0.757; therefore, we can guarantee that this set of tasks will always meet its deadlines.

The most common serious violation of the assumptions for Equation 1-2 is the independence of tasks. When one task can block another (resource sharing being the most common reason for that), then blocking must be incorporated into the calculation, as shown in Equation 1-3.

In this equation, B_j is the worst-case blocking for task j. Note that there is no blocking term for the lowest-priority task (task n). This is because the lowest-priority task can never be blocked (since it is normally preempted by every other task in the system).

Equation 1-3 Rate Monotonic Analysis with Blocking

$$\sum_{j} \frac{C_j}{T_j} + \max\left(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n\left(2^{\frac{1}{n}} - 1\right)$$

As mentioned above, it is necessary to take into account any chained blocking in the blocking term. That is, if one task can preempt a blocking task then the blocking term for the blocking task must take this into account. For example, in the model shown in Figure 1-5, the blocking term for HighPriorityTask must include the time that Low PriorityTask locks the resource plus the sum of the worst-case execution time for all the intermediate-priority tasks (because they can preempt LowPriorityTask while it owns the resource), unless some special measure is used to bound the priority inversion.

Detailed treatment of the analysis of schedulability is beyond the scope of this book. The interested reader is referred to [5,9,12], for a more detailed discussion of timeliness and schedulability analysis.

1.2.5 Modeling Performance

Performance modeling is similar to modeling schedulability. However, our concerns are not meeting specific timeliness requirements but capturing performance requirements and ensuring that performance is adequate. Performance modeling is important for specifying performance requirements, such as bandwidth, throughput, or resource utilization, as well as estimating performance of design solutions. The measures of performance commonly employed are resource utilization, waiting times, execution demands on the hardware infrastructure, response times, and average or burst throughput often expressed as statistical or stochastic parameters. Performance modeling is often done by examining scenarios (system responses in specific circumstances such as a specific load with specified arrival times) and estimating or calculating the system response properties. Performance analysis is done either by application of queuing models to compute average utilization and throughput or via simulation. This often requires modeling the underlying computational hardware and communications media as well.

One key concept for performance modeling is workload. The workload is a measure of the demand for the execution of a particular scenario on available resources, including computational resources (including properties such as context switch time, some measure of processing rate, and whether the resource is preemptible). The priority, importance, and response times of each scenario are required for this kind of analysis. The properties of the resources used by the scenarios that must be modeled are capacity, utilization (percentage of usage of total available capacity), access time, response time, throughput, and how the resources are scheduled or arbitrated. Chapter 4 discusses the UML Profile for Schedulability, Performance, and Time, and its application for modeling performance aspects.

PREVIOUS PAGE | TABLE OF CONTENT | NEXT PAGE

Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition)

ISBN: 0321160762
EAN: 2147483647

Year: 2003
Pages: 127

Authors: Bruce Powel Douglass

BUY ON AMAZON