Chapter 3

Reserve Model

This chapter gives a definition of reserves on resources, which form the basis of the reservation model. Reserves provide a framework for integrating admission control, scheduling, and usage enforcement. Issues in reserve management are also addressed.

3.1 Reserve abstraction

The reservation model defines the concept of a *reserve* against a particular operating system *resource*. A reserve is a first class object that represents a part of the capacity or a quantity of the resource that is set aside for a computation which presents that reserve along with a request to use the resource. The word “capacity” is used in a broad sense here; reserving a portion of the capacity of a resource means that a thread will have access to the resource subject to some detailed reservation parameters, and the parameters are specific to the resource and the reservation system implementation. As an example, a reserve might specify that 30 ms of computation time on the processor are reserved out of every 100 ms of real time.

Reserving resource capacity implies that the resource can be multiplexed among several computations, and the model focuses on multiplexed resources such as processors and network bandwidth. Other types of resources such as physical memory pages and buffers are not amenable to extremely fine-grained multiplexing, and these are referred to as discrete resources. In this model, discrete resources are reserved on a per unit basis and the reservation dedicates the resources units indefinitely rather than implying a multiplexed usage of capacity over time.

3.1.1 Reservation guarantee

A key requirement in offering a resource reserve abstraction in a system is that the reserved resource capacity be available, subject to the reservation parameters, to a computation which presents the reserve and requests the resource. If the system cannot guarantee that the resource capacity will be available as promised, the usefulness of the system is limited. Therefore, the enforcement of resource reservations is of critical importance. Enforcement is important not only to protect against malicious users, which may present a problem in systems where resources are shared by many, but also to relieve individual applications from the burden of ensuring that their own performance is strictly predictable and controlled. The system should tolerate applications which may try to use more than their reservation allows, isolating unrelated applications from this kind of behavior. This kind of *temporal isolation* is similar in concept to the isolation provided by a virtual memory system, which allows a process to try to access memory locations as it wishes, intervening only when a memory access is not allowed. In no case should a virtual memory system allow a process to access the memory of another process's protected memory, and likewise in no case should a reservation system allow a thread to impinge on the reserved resource capacity of another thread.

Thus, the system guarantees that resource capacity, given by the reservation parameters, will be available to the thread that has a reserve. However, it is up to the application programmer to make sure that the thread is in a position to take advantage of the resource capacity when it is made available. The reservation system itself makes no claims that a particular application will meet its timing constraints. For example, if an application blocks indefinitely waiting for a message, it may not be in a position to take advantage of resource capacity when it is available. For an application to have predictable real-time performance, it must have the proper resource reserves, and it must be able to use those resource reserves in a way that satisfies the timing constraints of the application.

3.1.2 Scheduling frameworks

The reserve abstraction can accommodate different frameworks for admission control, scheduling, and enforcement. Most of the features of reserves, specifically the operations available in the reserve abstraction, remain the same even if the scheduling framework changes. The primary differences in the interface to a different framework are the specification of the reservation request parameters for admission control and the resource usage statistics available from the enforcement mechanism.

3.1.3 Styles of programming with reserves

Reserves can be used in two different styles of real-time programming. Reserves support strict hard real-time applications and can equally well support more flexible soft real-time applications. The primary distinction between hard and soft real-time programming in the discussion of the reservation model is:

- whether resource usage requirements are carefully measured and specified in exact detail guaranteeing performance before the program is actually deployed (hard real-time), or
- whether resource usage requirements and resource capacity reservations are dynamically adjusted based on run-time usage measurements instead of being matched exactly during the design phase.

In either case, the resource reserves guarantee the requests that are admitted to the system, and whether or not those reserves are used for hard real-time programming or soft real-time programming depends on how the applications themselves use the reserves.

Another distinction in real-time programming using reserves is whether resources are reserved locally for each thread or whether they are reserved globally for an *activity* that may span multiple threads in different protection domains and even on different machines. In the activity-based model of using reserves, the originator of an activity acquires resource reserves for the activity and then passes those reserves along with any requests made to various servers. Using this model, accounting for resource usage across clients and servers is simplified, and the negotiation of quality of service parameters can be simplified as well. The reserves for each request come in with the request, and the server charges resource usage to those reserves when servicing the corresponding request. The responsibility of getting the appropriate resource reserves falls to the original client.

To summarize, several features of reserves are useful for both hard and soft real-time programming:

1. Take care of global admission control decisions, relieving the designer of doing global scheduling analysis.
2. Schedule threads on resources according to their reserves.
3. Accumulate usage information that could be useful during development, especially for adaptation in soft real-time applications.
4. Prevent interference from other real-time applications and non-real-time applications and activities that may be competing for the same resources.
5. And finally, reserves can serve either to separate the resource allocation and management of modules from each other or to integrate the resource allocation and management of modules, allowing reservations to span multiple threads and protection domains of a single activity.

3.2 Basic reserves

The *basic reserve* provides an abstraction for capacity on a particular resource. Note that this statement about basic reserves *does not guarantee that applications will meet their timing requirements*. The only guarantee is that *capacity will be reserved and available to be used*. We will explore the issue of what guarantees can be made at a higher level about the behavior of applications that use reserves.

The reserve itself is an operating system abstraction that is orthogonal to control structures like threads. A thread may bind to a resource reserve in which case the scheduler will use the information in the reserve when making scheduling decisions about the thread. Multiple threads may be bound to a single reserve, but typically a reserve will have only one thread bound to it at a time. The scheduler will always find an associated reserve, although sometimes that reserve will be a *default reserve* which has no actual reservation and just serves to accumulate the resource usage of all threads that make unreserved use of the resource.

The specification of the reservation depends on the type of resource. Multiplexed resource reservations include information about the amount of work to be done per period of real time. They may also include a delay requirement. This parameter would specify the maximum amount of time after the beginning of each period the thread will have to wait before getting its reserved time on the resource. Discrete resources reservations just specify a count of the units of discrete resource required; they include no notion of time.

Despite the differences between multiplexed and discrete resource reservations, they share the same basic structure. They both require:

1. a reservation specification interface,
2. an admission control policy,
3. a scheduling policy, and
4. an enforcement mechanism.

For discrete reserves like memory pages and network buffers, the reservation specification gives a number of units of resource being requested. The admission control policy for discrete resources would just check the availability of the requested number of units of resource. Since discrete reserves are by definition not multiplexed, they require no scheduling.

It is important to note that the basic reserve abstraction is independent of the admission control and scheduling policies used. For multiplexed resources, reserves provide a framework to request resource capacity reservations, do admission control, schedule computations, and enforce capacity reservations. The choice of admission control and scheduling policies will impact the way reservation requests are specified and the way the enforcement mechanism tracks their behavior, but the framework is general enough to accommodate different policies. The following sections illustrate the reserve model in terms of a periodic scheduling framework.

3.2.1 Reservation specification

The reservation system must provide a way for applications to specify the resource capacity they would like to reserve. The form of the specification differs from resource to resource, and different admission control and scheduling policies may require different reservation specification parameters. In the most general sense, reservations specify a duration of usage with some time constraints be available, used, and replenished by some specific regimen.

As an example of the kind of parameters that might appear in a specification, a resource reservation may specify an amount of time to be spent on the resource per period of real time, and it may specify a start time for the periods as well. For example, a reservation request might specify 30 ms every 100 ms starting at 1:00pm.

Figure 3-1 illustrates how the reserved time might be consumed in a simple computation time per period of real time framework for reserves. The reserved computation time is available to be used during each reservation period. The computation time is guaranteed to be available at some point during the period; it is not guaranteed to be in any particular place such as the front of the period or the end of the period.

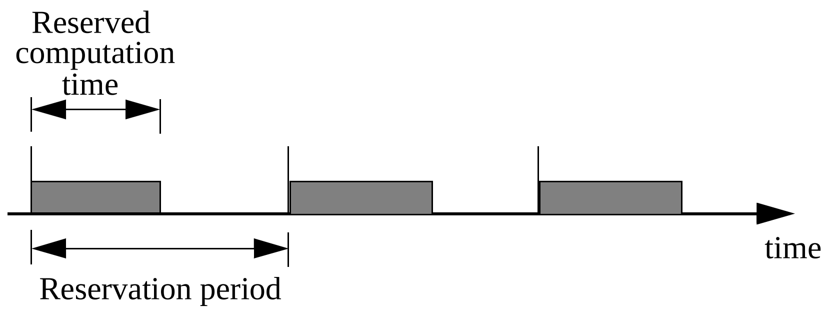

Figure 3-1: Periodic Scheduling Framework

There are many different scheduling policies and scheduling analysis techniques that could be used to provide a reserve abstraction, each of which would require a corresponding admission control test, scheduling policy, and enforcement mechanism.

3.2.2 Admission Control

An admission control policy associated with each resource decides which reservation requests for that resource can be admitted and which must be denied. It makes this decision based on the parameters provided in the reservation request and information about the other reservations that have already been granted for that resource. The admission control policy necessarily depends on the scheduling policy in order to do an admission control analysis.

3.2.3 Scheduling

The scheduling algorithm for a resource makes decisions about the order in which threads receive time on a resource. The scheduler looks at the reserve owned by each thread that is ready to run, and uses information in the reserve to determine which thread will get access to the resource. The algorithm supports the decision made by the admission control policy.

The scheduler must also coordinate with the enforcement mechanism to make sure that it does not try to schedule threads associated with a reserve that has already used its reserved resource time for a particular reservation period. Thus a reserve which still has time left on its reservation is in “reserved mode” and one that has run out of time is temporarily in “unreserved mode.” This represents a significant departure from other real-time scheduling algorithms, which generally assume that the resource usage requirements of application are accurately characterized and need not be enforced [67].

3.2.4 Enforcement

The reservation system must ensure that processes do not use more than their reserved capacity or reserved units of a resource. Enforcing reservations on discrete resources is straightforward; the system ensures that a resource dedicated to one process is not re-allocated to another process. Enforcing multiplexed reservations requires the system to keep accurate usage numbers that describe how much capacity has been consumed against each reservation. If a thread attempts to use some capacity beyond its reservation, the system must recognize this and actively prevent the process from consuming any additional capacity in reserved mode (consuming additional capacity in an unreserved mode may be allowed.)

If for some reason the reserved time on a resource is not used by the owner of a reserve in a given reservation period, that allocation of time is lost to the owner. The owner may not be in a position to use the resource if it is blocked waiting for some other resource to become available or for synchronization or communication with another computation. The resource will not necessarily be idle for that amount of time since the scheduling policy is free to allow an unreserved computation to use the resource (as long as the unreserved computation can be preempted to allow the reserve owner to use the resource). This implies that a computation may not save up reserved time (by not using it) and then use it all at once in a burst. The allocation of resource time is available during each period, but cannot be carried over past the end of the period.

On the other hand, if the thread that owns a reserve consumes the entire reserved allocation for a reservation period and attempts to continue executing, the thread will compete with the other unreserved computations for the resource under whatever policy the resource scheduler uses for unreserved computations. Thus a reserve may be in *reserved mode* where it still has some resource usage allocation left for the current reservation period, or it may temporarily be in *unreserved mode* where the allocation for the current reservation period is depleted (until the next reservation period).

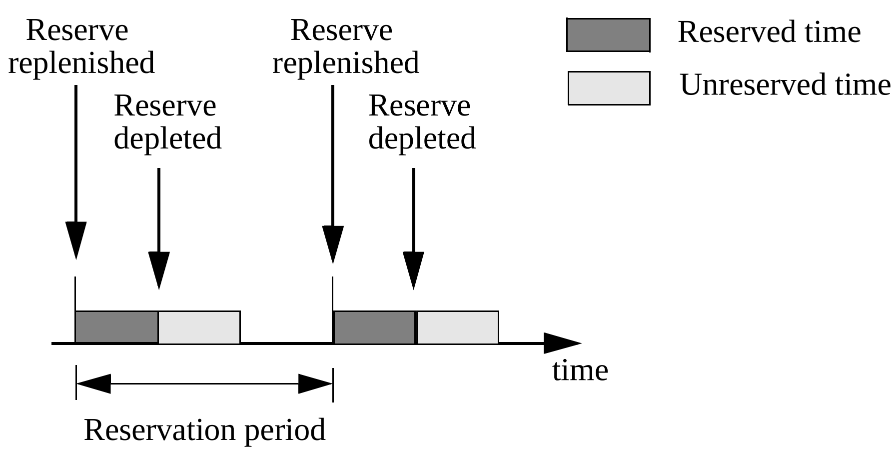

Figure 3-2: Enforcement Illustration

Figure 3-2 shows the reserved time made available on a particular resource for a single reservation. At the beginning of each reservation period, the allocation of reserved time is replenished, and the thread that has this reservation uses the resource in “reserved mode.” After the reserved time allocated for that period is depleted, the enforcement mechanism generates a scheduler event to indicate that the reserved time has been consumed for that period. The scheduler is responsible for using that information in making scheduling decisions. In the figure, the thread continues to use the resource in “unreserved mode,” consuming more time on the resource at the discretion of the scheduler. The execution history shown in the figure is based on the assumption that no other threads compete for the resource and that the policy lets it run in timing-sharing mode after its reservation has expired, and so the thread can get time in unreserved mode. At the beginning of the next reservation period, the reserve is replenished and the thread can run in reserved mode again. The main point of this figure is that the enforcement mechanism tracks resource usage and raises this “reserve depleted” event for the scheduler. The scheduler can then use this information in making future scheduling decisions. For example, it can give other reserved activities preference or allow unreserved time-sharing activities to use the resource.

Three important issues arise in the design of the enforcement mechanism:

1. how to accurately accumulate resource usage,
2. how to notice that a thread has depleted its reserve for a resource,
3. how to know when to replenish the allocation of a reserve.

To accurately accumulate the resource usage for each reserve, the system records the time during each reserve switch. A reserve switch occurs in two cases: when a thread context switch is performed or when a thread is changing the reserve against which it will charge its computation time. In the case of a thread context switch, the reserve switch records the current time, cancels the overrun timer (which signals the reserve depletion event as described below), computes the time the old thread ran, and adds that time to the accumulated usage of the old thread. The reserve switch mechanism then saves the current time for use later in computing the computation time of the new thread and sets the overrun timer. A reserve switch triggered by a thread changing the reserve to which it wants to charge its computation time works the same way, the only difference is how the reserve switch is triggered. Figure 3-3 illustrates how timers are used in enforcement. In the execution history on the timeline, it shows the reserved activity of interest in a solid pattern and some other activity (associated with other reserves) in a striped. The reserve switches (also context switches in this example) between these activities occur where the striped boxes and the stippled boxes meet.

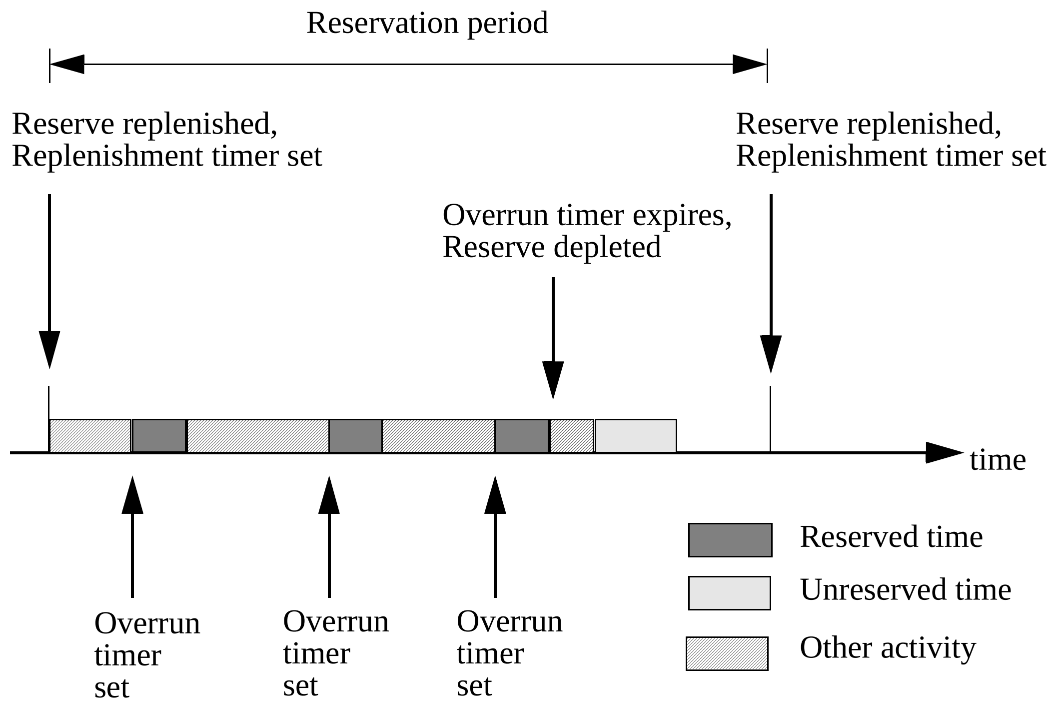

Figure 3-3: Enforcement Timers

The overrun timer is set during the reserve switch to expire at the end of the new thread's remaining reserved computation time or at the end of its reservation period, whichever is earlier. If the new thread is preempted before its reserved computation time is completed, the overrun timer will be cancelled. If the new thread consumes all of its reserved time, the overrun timer expires, and the scheduler is called to take some action based on that event. Figure 3-3 shows where the overrun timer is set for the reserve of interest; the overrun timer may also be set for the other activity if it is reserved, but that is not shown in the figure. The overrun timer in the figure does not actually expire until the last time it is set.

The handle replenishment, each reserve has a replenishment timer which is started at the reserve start time and which expires periodically at each reservation period. The replenishment timer records the usage accumulated on the corresponding reserve at the time of the reservation period and records that along with the current time as a "usage checkpoint." Then the timer handler changes the state of the reserve to reflect a new allocation of resource usage and resets the periodic replenishment timer. This replenishment model corresponds to a deferrable server [120] replenishment scheme; other replenishment methods are described and analyzed by Sprunt [111,112]. Figure 3-3 illustrates where the replenishment timer is set relative to the reservation period.

3.3 Reserve propagation

One important feature of the reserve model is that reserves can be passed from clients to servers, enabling the server to take advantage of the resources the client reserves for its entire computation. Passing reserves also enables the server to charge the resource usage it consumes to the appropriate client, preserving system-wide consistent usage accounting. This is called *reserve propagation*.

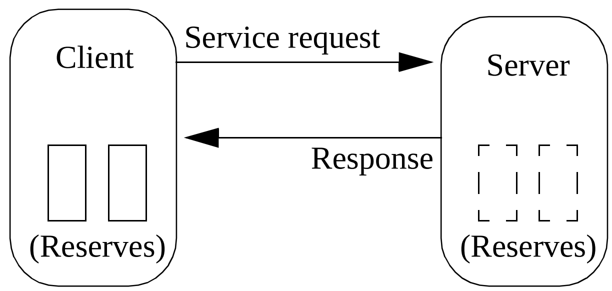

Figure 3-4: Reserve Propagation

Figure 3-4 illustrates a client/server interaction with reserve propagation. Assume that the client acquires resource reservations sufficient for the computation that it will perform locally as well as the computations to be done by servers on the client's behalf. In the figure, these reserves appear in the clients as two small rectangles. The interaction is a straightforward remote procedure call from the client to the server. For simplicity, assume that the client sends an RPC request to the server and waits for the reply. The server processes the request and sends an RPC reply, and then the client receives the reply.

When the client send the service request, it also sends references to the reserves that it has allocated. These reserves are to be used by the server as it processes the client's request. Thus, the server must start charging its resource usage to the client's reserves when it starts processing that request, and it must stop charging to those reserves when it finishes with the request and sends back the response.

Ideally, a server would schedule service requests to execute in the same order that the computations would execute if the clients could do them instead of the server. For example, the processor scheduler orders ready threads based on the processor reserve parameters. This ordering can be seen as a sort of "priority" ordering among those activities. If a thread makes a request of a server, the server should take the "priority" in the scheduler's ordering while it is servicing that request. Then the fact that a client relies on a server for some computation does not affect the progress of its computation with respect to other threads.

To help the server achieve this ideal, the RPC mechanism should propagate the reserve "priority", as represented by the reserve and its reservation parameters, of the client to the server. The queue of service requests for the server must be maintained in reserve "priority" order, and a kind of “priority inheritance” must be used to prevent priority inversion in the access to the server.

On the receipt of a new service request, his “reserve priority inheritance” mechanism enqueues the request in the priority queue, and then it checks to see if the thread that will service the requests is waiting for new requests or servicing a previous request. If the thread is busy and the “priority” of the new request is greater than the “priority” of the currently processing request, the RPC mechanism sets the priority of the thread so the priority of the new request. It does not, however, change the reserve that the thread is charging against. When the server thread finishes the previous request and receives the new request, it keeps the priority of that new request (which it inherited before) and also begins charging to the reserves associated with the new request. When the request is finished, the server thread stops charging against the client’s reserves.

The “priority” inheritance mechanism described here, which is referred to as “reserve propagation”, differs from traditional priority inheritance in two ways:

1. reserve propagation specifies how a server should change the reserves it charges to based on the client it is servicing whereas traditional priority inheritance has no notion of charging to a reserve or account,
2. the “priority” of the server may change during the course of the request processing if the reserved resources are depleted during that time whereas with traditional priority inheritance, the priority is fixed.

The fact that a server’s “priority” may drop during request processing complicates reserve management and reserve “priority” inheritance. When a reserve is depleted, the scheduler must determine whether the thread charging against the reservation inherited the reserve “priority” or not. If not, the thread’s order in the ready queue may change. If so, the scheduler must determine from the threads pending request queue what the appropriate reserve “priority” for the thread should be, given the change in the state of the reserve that was depleted.

Reserve propagation from client to server is not mandatory. The next chapter discusses different programming models where this is useful and where it is not. Briefly, reserve propagation is useful when the system is organized such that an application allocates the resource reserves it will need for all of its activities and passes those reserves around to the servers it invokes to do work on its behalf. In this model, applications need not negotiate quality of service parameters explicitly with these servers. The scenario where reserve propagation is not that useful is where the system is organized such that applications negotiate quality of service explicitly with all of its servers.

3.4 Example scheduling frameworks

Many different admission control and scheduling policies could be used to support the reserve abstraction. For example, reserves based on rate monotonic scheduling [67] would be able to guarantee the availability of a certain amount of time on a resource per period of real time. For pure rate monotonic scheduling, the delay associated with receiving the computation time in each period would be the length of the period itself. For deadline monotonic scheduling [64], the delay bound could be shortened. The following sections discuss these frameworks and others in more detail as they apply to the reserve abstraction.

3.4.1 Rate monotonic

The rate monotonic (RM) priority assignment of Liu and Layland [67] can guarantee that a certain amount of computation time will be available for a reserve for each period of real time with a delay bound equal to the length of the period. The discussion of rate monotonic scheduling uses the word "task" to denote the series of instances of a computation; with reserves, it is understood that this computation represents available resource capacity and not necessarily a complete program. Under rate monotonic scheduling, higher priority is assigned to the higher frequency programs. The rate monotonic scheduling analysis yields a basis for a processor reservation admission policy.

3.4.1.1 Reservation parameters

Reservation parameters for the simplest form of rate monotonic based reserves include computation time and reservation period: A start time is also useful for controlling the phase of the periodic reservations. This allows the programmer to synchronize the availability of the reserved computation time with a periodic program.

3.4.1.2 Admission control decision

Let n be the number of periodic tasks and denote the computation time and period of program i by C_i and T_i, respectively. Liu and Layland proved that all of the tasks would successfully meet their deadlines and compute at their associated rates if

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

When n is large, $n(2^{1/n} - 1) = \ln 2 \cong 0.69$. This bound is pessimistic: it is possible for task sets which do not satisfy the inequality to successfully meet their deadlines, but this cannot be determined from the Liu and Layland analysis.

An admission control policy follows naturally from this analysis. To keep track of the current reservations, the system must remember the utilization of the tasks that have reserved computation time, and the total reservation is the sum of these individual utilizations. A simple admission control policy is to admit a new reservation request if the sum of its utilization and the total previous reservations is less than 0.69. Such a policy would leave a lot of computation time that could not be reserved. One possibility is to use that time for unreserved background computations. Another possibility is to use the exact analysis of Lehoczky *et al.* [63] to determine whether a specific collection of reservations can be scheduled successfully, although the exact analysis is more expensive than the simple, pessimistic analysis above. In their work, Lehoczky *et al.* also gave an analysis showing that on average, task sets can be scheduled up to 88% utilization. So in most cases, this unreservable computation time is only 10-12% rather than 31%.

It should be noted that the rate monotonic scheduling algorithm was analyzed under simplifying conditions. Liu and Layland [67] made the following assumptions to enable their analysis:

- arrivals are periodic, and the computation during one period must finish by the end of the period (its deadline) to allow the next computation to start,
- the computation time of each program during each period is constant,
- computations are preemptive with zero context switch time, and
- computations are independent; i.e. computations do not synchronize or communicate with other computations.

In the context of the reserve abstraction, this means that rate monotonic scheduling can be used to guarantee that resource capacity is available to the applications. However, applications that have precedence constraints with other applications may not be in a position to use the available resources.

3.4.1.3 Scheduling

Reserved mode activities get precedence over unreserved. Among reserved mode activities, smaller period gets precedence over larger. Among unreserved activities, some time-sharing algorithm may be in effect.

3.4.1.4 Enforcement

Accumulate usage in each period. Update usage (determined using accurate measurement techniques) at each context switch. Set a timer for the currently running activity to expire at the end of its reserved usage. Set another timer for each reservation period.

3.4.2 Deadline monotonic

The deadline monotonic scheduling (DM) algorithm [7,64,66] is closely related to the original rate monotonic (RM) algorithm [67]. DM has the same kind of periodic scheduling frame as RM; the difference is that with DM, there is an additional parameter called the deadline specifies the duration of time by which the computation released at the beginning of the period must be completed. For the original version of RM, this deadline is assumed to be the end of the period, when the next instantiation of the computation will be released. For DM, this deadline is specified explicitly.

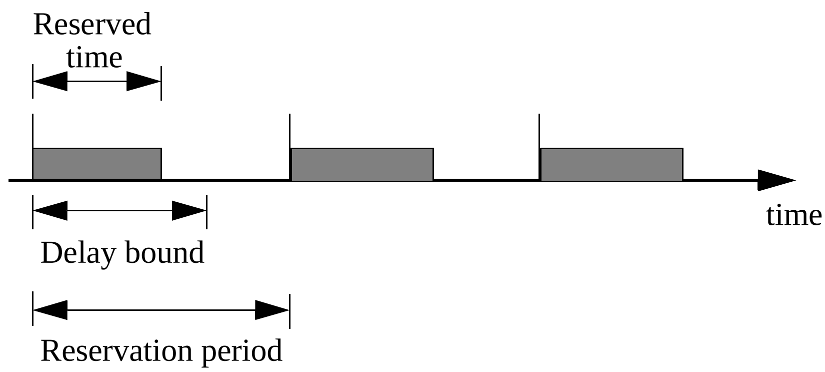

Figure 3-5: Deadline Monotonic Scheduling Framework

Figure 3-5 illustrates the periodic scheduling framework of DM along with the additional deadline parameters that does not appear in the original RM algorithm. The deadline in this case is before the end of the period, but it could also be after the beginning of the next period (in which case there would be multiple instantiations of the computation, started in different periods, at one time).

3.4.2.1 Reservation parameters

The reservation parameters for DM are the same as RM with the addition of the deadline parameter. As an example, a reservation request may specify 30 ms on the resource be reserved for every 100 ms with the delay constraint that the 30 ms must be available within 50 ms of the beginning of each 100 ms period. As for rate monotonic scheduling, a start time parameter is useful for synchronizing reservations with periodic threads and with other reservations.

3.4.2.2 Admission control decision and scheduling

Schedulability analysis tests for DM are given by Lehoczky [64] and by Audsley et al. [7]. These tests are quite a bit more complicated than the simple schedulability bound test for RM, involving systems of equations that have to be checked. Even so, the schedulability tests provide suitable admission control decisions for a reservation mechanism based on DM.

Scheduling is based on the deadline monotonic algorithm that assigns fixed priorities to tasks based on the deadline value. Shorter deadlines are assigned higher fixed priorities than longer deadlines. As with the rate monotonic algorithm, the reservation mechanism distinguishes between reserved mode activities and unreserved mode activities. At the beginning of any given reservation period, an activity with a reservation is in reserved mode until it consumes all of the reserved time for that period. It is then changed to unreserved mode. The scheduler services reserved mode activities first, in order of their deadline values. If there are no reserved mode activities, unreserved activities are scheduled.

3.4.2.3 Enforcement

The enforcement mechanism accumulates usage in reserved mode until one of the following occurs: the resource usage reserved for that period is consumed or the deadline time has passed. In either case, the activity is changed from reserved mode to unreserved mode where it can compete with time-sharing activities for the resource.

3.5 Basic reserve types

Operating systems manage many different kinds of resources that system and user programs may use to do their work. The most important examples are processors, physical memory, buffers, and network bandwidth.

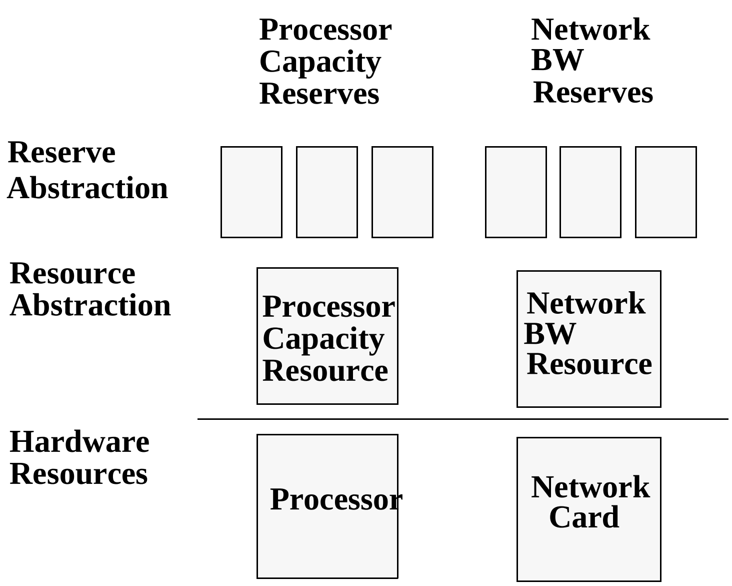

Figure 3-6: Resources and Basic Reserves

Many of these resources must be managed in the reservation system, so we define *basic reserves,* which are used to reserve and control the usage of different types of system resources. Each basic reserve type is associated with a resource type in the system. Figure 3-6 illustrates the relationship between the operating system resources and the basic reserves.

3.5.1 Processor

Processor capacity reserves represent reserved time on a processor. Reserve requests specify capacity in terms of time that will be reserved on the processor, rather than in terms of instructions that will be executed or any similar measure of processor usage. The requests may specify other information, depending on the admission control and scheduling policy in effect. The discussion in the following sections assumes a deadline monotonic scheduling framework where the reservation request specifies the amount of time to reserve on the processor, a period at which the allocation will be replenished, and a delay bound. These sections will cover these topics in more detail and will additionally discuss issues in enforcement, blocking time, and usage monitoring.

3.5.1.1 Units of work

Processor reserves deal with allocating real time on a processor rather allocating a sequence of instructions. The reason for this is that reserving instructions would be too difficult. It would require knowing the exact sequence of instructions to be used with the reservation, fixing the exact sequence for accuracy (to avoid cache effects, etc.), allocating a time slot on the processor to execute those instructions, etc.

Several pitfalls complicate the use of this reservation model that is based on time spent on the processor. For example, DMA can impact the amount of work that gets done in a certain amount of time spent on the processor. Cache effects can introduce variance in the amount of work per time on the processor. Processor pipeline flushing at context switches decreases the amount of work done during a fixed time on the processor. These are all second order effects, but their impact should be accurately characterized.

Processor reserves leave it to the individual applications and other higher-level software to make an appropriate mapping between the computational requirements of the applications to the appropriate reservation specification. For hard real-time applications, accurately characterizing processor requirements is very important. For soft real-time applications, an adaptive approach is the key to dealing with the fact that reservations are for time on the processor rather than work done by the processor. These applications can look at their own behavior and make adjustments as necessary.

3.5.1.2 Admission control and scheduling

For processor reserves in a rate monotonic scheduling framework, a reservation request consists of three parameters: a computation time, a period, and a start time. The admission control and scheduling policies described here are based on rate monotonic scheduling [67] as described above.

3.5.1.3 Enforcement

The enforcement mechanism for processor reserves must keep track of the processor usage for each reserve so that a scheduling event can be raised at the point where the reserves allocation has been depleted for a given reservation period. The usage measurement task is complicated by the fact that a thread charging to a particular reserve may be preempted, and so at each thread context switch, the usage numbers must be updated to reflect the usage since the last context switch.

Accurately accumulating resource usage

To accurately accumulate resource usage in the face of preemptive use of the resource, the system must, at each context switch, compute the usage since the last context switch. This can be achieved by recording the start time of the computation (at the last context switch) and then computing the difference between the time at the current context switch and the time of the last context switch. This is the time the last thread was using the processor resource, and this time is added into the usage accumulator for that thread's reserve. Thus the accumulator keeps an accurate account of the resource usage charged to it.

Noticing reserve depletion

The enforcement mechanism must be able to notice when the reserve of the currently executing thread becomes depleted. To do this, the system at each context switch computes the longest contiguous time the thread is entitled to execute on its reserve, and it sets a timer for that time. If a context switch occurs before the timer expires, the accumulators are updated and the timer is set for the next thread to execute. If the timer expires while the thread is executing, the system updates the accumulators, marks the reserve as "inactive", and calls on the scheduler to make some decision based on the new state of that reserve.

Replenishing a reserve's allocation

Each reserve must have its allocation replenished at the beginning of each reservation period so that the time on the resource is available if it is needed during that period. To do this, the system uses a periodic timer for each reserve which is set to expire at the beginning of that reserve's reservation period. When the timer expires, the state of the reserve is updated to reflect a full allocation of resource usage for the upcoming period.

3.5.2 Physical memory

A physical memory reserve represents a collection of physical memory pages. Physical pages are discrete resources, so they support simple discrete reservations. The more interesting question is how the owner of a page reserve uses this collection of physical pages. Basically, pages can be locked down or paged in and out, and they can be prefetched or demand paged. A small application which could fit into its page reserve would benefit from prefetching its image into the page reserve and locking down the pages. A larger application might benefit from prefetching and locking down some (more frequently used) pages while keeping other physical pages available for less frequently used logical pages to be paged in and out. The advantage of using physical page reserves for these larger applications is in the increased control reserve give the application over traditional time-sharing demand paging replacement policies. With physical page reserves, the owner of a page reserve will at least be isolated from competition for pages with other threads in the operating system.

3.5.2.1 Admission control and scheduling

The admission control policy for this discrete resource is as follows: if there are enough free physical pages to satisfy a reservation request, then the reserve is granted; otherwise the reservation request is denied. The physical page reserve contains the number of pages that are reserved, and these pages are completely free and ready to be used by the thread that owns the reserve. The system may want to keep some number of physical pages as "unreservable" pages to allow time-sharing threads enough resources to make progress.

There is no scheduling of the use of pages by the reserve mechanism.

3.5.2.2 Enforcement

The enforcement of reservations for this discrete resource is relatively straightforward: a thread that has a physical memory reserve can use pages in its own memory pool and can also use pages from the time-sharing free page pool. Thus a thread using a physical memory reserve is assured of having at least the reserved number of pages available and possibly more. At no time will the pool of pages in the reserve fall below the reserved number.

3.5.3 Network bandwidth

Reserves for network bandwidth represent reservations for time on the network device. The system must include a mechanism for identifying the reserve to be used for incoming network packets. These reserves will typically be closely coordinated with processor capacity reserves and with bandwidth reservation supported by the network. The operating system will control the amount of outgoing traffic for each session (or virtual channel), and it will ideally coordinate with a network reservation system to limit the amount of incoming traffic for each session.

3.5.3.1 Units of work

The unit of work for a network bandwidth reserve is the transfer of a number of packets of a particular size (which will probably be constant, the MTU). Servicing of single packets is certainly non-preemptive, and it should also be possible to bundle multiple packets into non-preemptive work units.

3.5.3.2 Admission control and scheduling

The reservation specification for net bandwidth reserves includes a reserved time per period of real time, and possibly an indication of expected blocking time.

Timeline or rate monotonic scheduling frameworks among others would be appropriate for net bandwidth reserves. Several important issues relate to the non-preemptive nature of the work unit. Ideally, the expected blocking time would be used in the admission control policy and scheduling algorithm.

3.5.3.3 Enforcement

Accurate measurements of usage time can be computed between requests. This information can then be used in the enforcement mechanism and for input into scheduling policy decisions.

3.6 Reserve management

3.6.1 Default reserves

Default reserves exist in the system to simplify the implementation of the reservation mechanism by providing “reserves” for non-real-time programs to charge usage against. These default reserves do not actually represent reserved resources, but they do accumulate usage for all activities that have created their own reserves or had reserves created for them.

For example, new threads are assigned to run under the default processor capacity reserve when they are created. Thus a thread will charge its time to this global reserve until it acquires a reserve of its own.

3.6.2 Composite reserves

Having many types of reserves allows flexibility in specifying resource requirements to the system and in allocating resources, but the job of managing those resource reservations at the user level becomes more involved. For example, a multimedia application, such as a video player, might reserve resources for several constituent activities. It might reserve some processor capacity for the module which reads audio and video data from the disk and passes the data to an audio server and a display server. It might also reserve processor capacity for a control module which provides fast response to interactive control commands from the user. The player part might also reserve physical memory and message queue buffers at the file system manager. Each of these reservations has an associated reserve, and we would like to be able to collect a subset of these reserves under a single name to avoid having to refer to them individually.

Grouping related reservations together helps alleviate this complexity. The model allows reservations for different types of resources, and the situation arises where a program has reserves for several different resources. Since it has to present the appropriate reserve handle to be able to use a resource, a way of grouping all of the reserves under one handle would make it easier for the program to identify its reservations to the system and to the servers it invokes.

A *composite reserve* groups reserves for different resource types under a single handle. A composite reserve has the following properties:

- it will contain a number of basic reserves,
- it may contain only basic reserves (no composite reserves),

• it may contain at most one reserve for each basic resource type

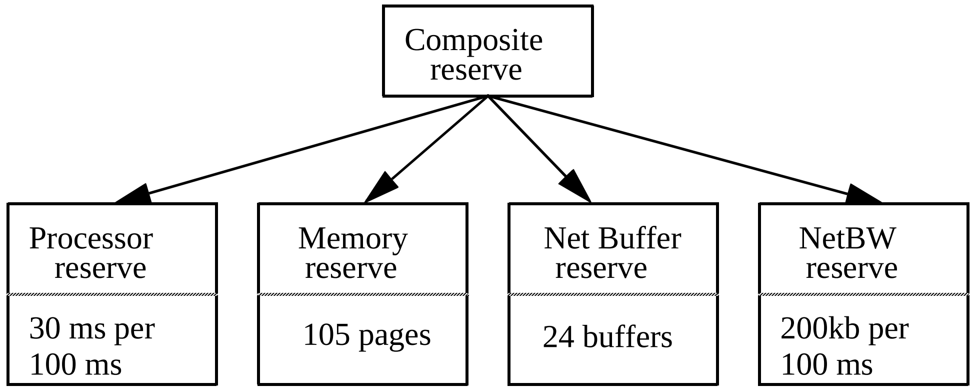

Figure 3-7: A Composite Reserve

Figure 3-7 shows the relationship between a composite reserve and its constituent basic reserves. In the video player example above, we might collect all of the reserves to be used by the player part (processor, physical memory, and message queue buffers) into a composite reserve. Then the system could use this reserve to reference the collection of resources reserved for the player. To charge computation time to the player, the system would take this reserve and look for the processor reserve under it.

3.6.3 Reserve inheritance

When a process creates a child process, the reserve of the parent is passed to the child, and the child runs against the resources reserved in the inherited reserve. This feature provides a way to allocate resources for non-real-time activities that create large process trees (like “make”). Reserve inheritance is appropriate for the automatic propagation of reserves for non-real-time programs, but real-time programs should generally configure their resources reserves explicitly. Figure 3-8 shows the difference between a process P whose children do not inherit its resource reserves and another process Q whose children and other descendants do inherit its resource reserves.

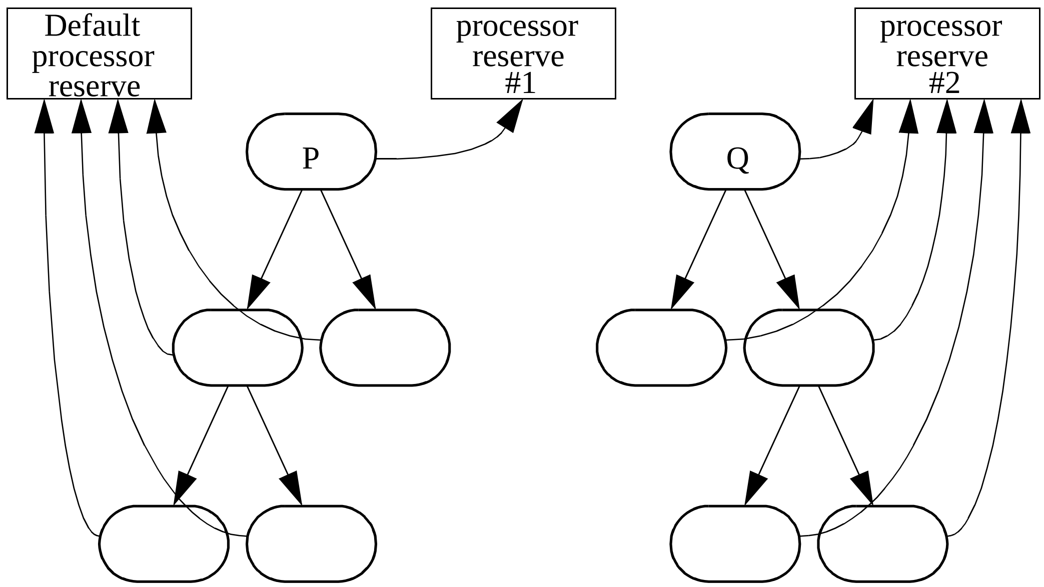

Figure 3-8: Reserve Inheritance

3.7 Chapter summary

This chapter described the basic reserve abstraction including reservation specification, admission control, scheduling, and enforcement. The idea of reserve propagation where a client hands reserves to a server to which it sends a request for service is shown to be a powerful mechanism for making reservations on a per-activity basis (rather than a per-thread basis). Several different scheduling frameworks which could be used in an operating system supporting the reserve abstraction were described, and the chapter discussed several different types of basic reserves for various resources such as: processor time, physical memory, and network bandwidth. A section on reserve management described default reserves, composite reserves, and reserve inheritance which address some practical issues in using reserves in a real system.