Chapter 5 - Resource Management

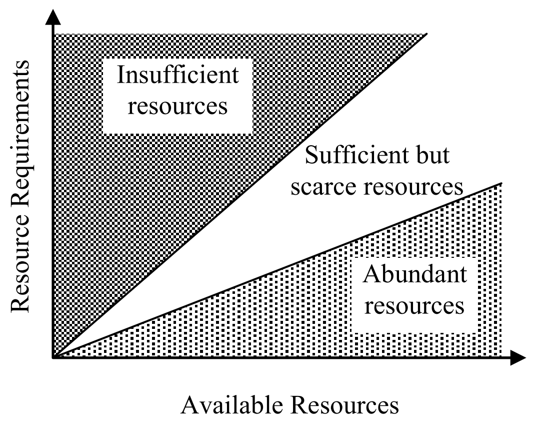
As we stated in Chapter 1, multimedia computing and networking introduce a number of problems for computing and networking systems. These include:

- Resource requirements for transmission and storage of multimedia: As we have seen in the last chapter, video compared with their textual counterparts generates a large amount of data. As computing systems continue to move towards supporting high definition and immersive environments, this trend will continue to grow.
- Continuity requirements: With audio and video having a temporal relationship between the data elements over time, large gaps in playback lead to less than desirable user experiments. The crux here is that a system that supports such media needs to be aware of such continuity of resource requirements.
- Burstiness of resource requirements: As we have seen in the previous chapter, compression of multimedia data, results in variable resource requirements over time. For systems that are managing the resources for applications that use multimedia, this means that they will need to be able to manage such variability over time.
- Synchronization: Multimedia presentations may require the coordination among multiple entities. For example, a multimedia presentation may require that text and animations synchronized with video (e.g. application tutorials). The other obvious example is coordinating audio and video.

The ability to provide resources to a multimedia application depends upon the amount of resources available to the amount of resources needed. Examples of such resource management exist everywhere in our everyday lives. For example, getting a reservation at a restaurant that is empty with lots of servers is a lot easier than getting a reservation at a full restaurant with few servers. To understand the relation between processing needs and resources, we describe the *Window of Insufficient Resources*.

5.1 Window of Insufficient Resources

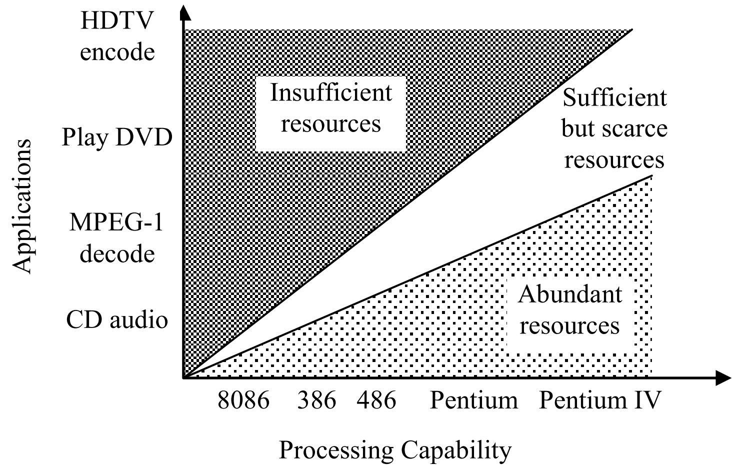
The *Window of Insufficient Resources* describes the relation of resource needs to resource availability. In its abstract version, there are resources and resource requirements. The "window" can then be broken into three distinct areas: Insufficient resources, Sufficient but scarce, Abundant. This concept is graphically shown below:



The resource requirements consist of the amount of resources an application or task needs to complete. Examples of resource requirements include playing a CD-audio disk, playing an MPEG-1 stream, or decoding and displaying a DVD. The available resources are the amount of resources available for the set of tasks that need to be completed. For example, if the resources are CPU cycles, the available resources might describe the various processors that are available. This might include CPUs such as the Intel 8086 processor, the Pentium process, or the Pentium IV processor. For a networking scenario, the available resources might include the amount of bandwidth available. For any given pair of processing requirements and available resources, the situation of resource requirements to available resources can be described as insufficient, sufficient but scarce, or abundant.

As another example, consider a restaurant. The available resources in this case are the number of servers and tables available in the restaurant, assuming that the food can be prepared at a much quicker rate than it is consumed. The "tasks" requiring resources are the number of customers. The goal of the restaurant is to serve as many people as possible, and if the wait becomes too long, to turn away people before they get upset (possibly handing them a gift certificate for a future visit). If the restaurant has 100 tables and two customers, there are abundant resources relative to the amount of customers that need to be served. If there are 100 customers and two tables, there will probably be no way to serve the 100 customers in sufficient time before they start leaving to other restaurants. Finally, in the case where we have 70 customers and 50 restaurants, some customers may need to wait a little while before being served, but ultimately they can be served without unduly being delayed. This simple example also provides insight into the relation between consumers and industries that supply services. As a consumer, I would like the restaurant to have 100 tables with two customers because it means that I will be served quickly. On the other hand, the restaurant owner would like to have few tables with many customers. This ensures that the restaurant owner does not have to have an excessively large retail space and an excessive amount of servers who would, otherwise, sit idle. Thus, in the restaurant case, a happy medium is achieved by having some people wait to eat while having most of the tables reasonably well utilized.

As another example, consider multimedia processing (resource consumers) and processor capability (resource capability). A representative translation of requirements to resources is shown below:



For DVD playback, insufficient resources existed on processors such as the 386 processor to provide continuous playback. Once processor speeds neared 800 MHz, there were sufficient resources to playback the DVD. The processor, however, was completely full and could be doing nothing else. Running an additional task on the CPU would cause a hiccup in the playback of the DVD. Finally, as processor speeds continue to increase, we are nearing a point where the CPU can easily provide DVD playback while doing other tasks such as word processing or web surfing.

All the above scenarios provide us with insight into how to handle such situations with multimedia systems.

Abundant Resources

With abundant resources relative to the resource requirement, providing service for the task is relatively easy. There are two ways to provide "abundant" resources to an application. This approach tries to *over allocate* (or *over provision*) resources so that the resources appear to be abundant. In the restaurant example, this could be providing 500 tables when the expected number of tables needed at any point is around 50.

The key advantage of an over allocation policy is that resource management becomes somewhat easier. There are always sufficient resources available to the application so that it does not need to do anything special. The key disadvantage, however, is that it requires resources to be allocated in a sub-optimal way. Clearly, reserving CPU cycles (or tables in our restaurant example) that ultimately do not get used wastes resources.

Scarce but Sufficient Resources

With scarce but sufficient resources, managing resource usage becomes more critical. There may be sufficient resources most of the time, on average, but there are times where the resources required exceed the available resources. In the restaurant example, there are times where the restaurant is extremely busy and times where it is nearly empty. For those customers (tasks) that need to finish in a prescribed amount of time additional resource management may need to be employed.

For computing systems, managing scarce but sufficient resources requires: (i) explicit management of resources, (ii) buffering and delay to smooth out resource availability variations, or (iii) a combination of (i) and (ii). For (i), a system or network can explicitly manage the amount of resources available so that the application requiring service can be handled without significant delay. For the restaurant example, there may be a set of 10 reserved tables for the 10 most important guests at the restaurant, while the rest of the customers share the rest of the tables in the restaurant. Thus, for the important guests, the availability of tables appears to be plentiful. In this case, there has been explicit management of the resources (tables). For (ii), the goal of the computing system and the restaurant are to handle as many tasks as possible. One way to deal with variability in resources is to use buffering and delay to manage the resource. In the restaurant example, a waiting area is commonly provided for perspective consumers. The idea here is that the variations in resource availability (i.e. the tables) can be smoothed out by having "extra" consumers around. Of course, this adds delay to the consumer of the resource.

The key advantage of explicitly managing resources is that the overall resources of the system can be maximized, reducing the cost or number of resources required to support the application. The key disadvantage of explicitly managing resources is that there is added complexity necessary to support managing the resource. For example, the system will need to keep track of which task has consumed how much of each resource that it needs. Further complicating the matter is that the application will need to know how much resource it actually needs. In practice, this might be extremely difficult.

Buffering to smooth out system resource availability is easier to manage than explicit management. It also allows a system to reduce the peak resource requirements of an application that may have varying resource requirement over time (e.g. the transmission of variable-bit-rate video). There are, however, a number of implementation concerns with buffering. First, buffering requires extra memory in a particular resource. Second, buffering can only smooth out the resource constraints to a certain degree, beyond which, the task may fail. Third, it *typically* adds delay to the system. The reason we italicize typically is that in some applications the delay, if appropriately planned for is never seen by the application. For example, retrieving data ahead of time off of a disk can be accomplished without actually delaying the playback.

Insufficient Resources

If insufficient resources exist to accomplish a task, then the only alternative to making the task possible is *dropping* some of the resource requirements. For example, DVD video decoding became nearly possible on 600 MHz Pentium processors. Depending upon other background tasks running on the system, the processor may or may not be able to decode the video. If the processor could not keep up, then not decompressing some of the frames sheds CPU resources. In most early DVD players, this was accomplished by dropping B-frames. B-frames, while providing the highest compression ratios, required the most amount of memory and computation to decompress. Therefore, completely skipping these frames saved many CPU cycles.

5.1.1 Summary of the Window of Insufficient Resources

The Window of Insufficient Resources presents a framework, by which, one can think about how resources are managed and how to make continuous tasks such as multimedia runnable on systems. As we will see in the next several chapters, tasks may move between having insufficient and abundant resources during the lifetime of the application. For example, in an Internet environment, the amount of resources (bandwidth) available to the application will continue to vary over time, causing the application to move between periods of abundant resources and insufficient resources). Finally, it should be noted that the various techniques described in this section can be combined in different ways to achieve high-quality continuous media applications.

5.2 Managing Resources

In order run multimedia applications, the resources need to be explicitly managed, or the application needs to adapt the resource requirements over time. In the former, the system needs to have mechanism to achieve such management. This management typically manifests itself as *Quality of Service*. Quality of Service (QoS) is typically a measure of how well a system is providing a service. Without specific metrics to measure, QoS is a meaningless topic. These metrics are often measured with respect to the application's requirements. There are several distinct phases of resource management including quality of service specification, schedulability testing, QoS calculation and resource reservation, and scheduling/delivery.

5.2.1 Quality of Service Specification

Each multimedia application needs to specify to the underlying system *what* its resource requirements actually are. Without it, it is impossible to manage the system. QoS parameters inform the system of the tasks' requirements. For the restaurant example that we have been using, a QoS specification for a customer would probably include the number of people needing a seat and the time that they are willing to wait. The restaurant host could then decide whether or not the restaurant can handle that customer.

For an operating system, the quality of service specification usually includes the % or amount of CPU cycles needed. It also usually includes the rate at which the CPU cycles are needed. For example, if a DVD application declares that it needs 50% of the CPU. Having one continuous second of CPU *every other* second would probably useless. Having 10 milliseconds every 20 milliseconds (or 50% of the CPU every 20 milliseconds) might be a more appropriate QoS specification

For a network, the QoS specification can include a large number of parameters including throughput, delay, jitter (the amount of variation in end-to-end delay), packet loss, reliability, etc. We will discuss these in more detail in a couple of chapters.

5.2.2 Schedulability Testing

Once an application has specified its QoS specification, the system needs to determine if sufficient resources exist to accomplish the task. This is the *schedulability test*. If sufficient resources exist, the application can either choose not to run or modify its QoS specification. In the restaurant example, the customer might have a QoS specification of 4 people and 15 minutes. The restaurant host, examining the people currently at tables may determine that it will take 20 minutes. The customers at this point can either choose to update their QoS specification or can decide to go to another restaurant. This scenario highlights two of the key problems in delivering QoS. First, it is hard to determine the needs of a task ahead of time. For example, the QoS specification is great for the customer but lousy for the restaurant. In order to make it easy for the restaurant to accept customers, it really needs to know *how long* the customers will need the table for. For example, there might be two sets of customers that have a 2 person, 15 minute wait QoS specification. The first set might just want dessert, while the other set may want an appetizer, entrées, dessert, and coffee. The second problem raised by this example is that it might not be able to determine how much resources it needs ahead of time. Some of the specification may be dependent upon the underlying resource. In the restaurant example, the customer has no idea how long it will take the chef to cook their meals. In addition, the chef will not know ahead of time what they are planning on ordering. Their meal may be easy to cook or might be extremely time consuming.

In most computing systems, the QoS schedulability test (and QoS specification) is typically accomplished in a statistical manner. For example, for a network connection, the network may specify that the network will be "on" for 95% of the time. Moving the network to 100% may require significant additional resources (and cost) to accomplish. Other statistical parameters can include packet loss %, etc.

The schedulability test is meant to take a *complete* QoS specification for an application and determine if sufficient resources exist to deliver on the QoS specification.

5.2.3 QoS Calculation and Reservation

Once it has been determined that a particular multimedia task can be scheduled on a system, sufficient system resources may need to be set aside to make sure the task meets the application's requirements. The QoS calculation converts the QoS specification into resource-centric requirements. The QoS specification may directly specify the resources (e.g. network bandwidth), but may also specify a higher-layer specification that needs to be mapped into a resource-centric requirement. For example, in playing back a DVD, the QoS specification may include the bit-rate of the video and the frame rate required. The underlying system will reserve CPU resources to accomplish the task. In addition, it may also convert the QoS specification into reservations for memory and disk bandwidth, something that the application may not have any knowledge of.

Once all the resources have been determined, the resources need to be reserved (set aside) for the application to use.

5.2.4 Resource Scheduling

The previous steps in resource management are required to be done once when the application starts up (or when the application requires a change in its QoS specification). The *resource scheduling* step is the step that actually carries out the QoS specification. There are two aspects to resource scheduling: delivering the resource and policing the resource.

Delivering the resource is the system meeting the reservation that it has accepted. In the restaurant example, this means that if it has accepted a 2-person, 15-minute wait specification, the customers will be seated in that time frame. For computing systems, this involves delivering the reserved CPU cycles. For networking systems, this involves delivering the data packets in a manner that meets the QoS specification.

As previously alluded to, one of the main problems with resource reservations is that the amount of resources needed may not be known ahead of time and a best guess will be made. For example, in the DVD scenario, the application may specify that it has an average requirement of 5 Mbps with a peak requirement of 8 Mbps. What happens if the DVD player, for a very short time, requires more than 8 Mbps? The system can *police* the application to make sure that it is adhering to the resource requirements and QoS specification that it originally gave to the system.

From the application perspective, this might also involve *checking that the system is providing the resources* that it accepted to deliver.

5.3 Chapter Summary

Multimedia applications are hard to support because of their larger resource requirements, their continuity requirements, the burstiness in their resource requirements, and their synchronization requirements. The window of insufficient resources provides us a framework in which to think about how resources are delivered and how application can adapt to deal with insufficient or scarce resources. Finally, either explicit resource management or application adaptation are necessary. Explicit resource management provides harder guarantees on service at the expense of additional system complexity and potential underutilization of resources. Application adaptation makes system building easier, but requires the application to be more complex to deal with resource inavailability.

Chapter 6 - Real Time Systems

In order to provide resource guarantees to an application, the system needs a mechanism in which to provide the resources that it has reserved for the application. Obviously, over allocating resources can help minimize the effect of bad resource scheduling. The question is, however, how much over allocation is needed? Furthermore, what happens when multiple applications require processing at the same time. In order to help understand how to support such resource guarantees, we will describe real-time systems in this chapter. Real time systems mechanisms provide the basis of many real-time systems.

In the remainder of this chapter, we provide an overview of real-time systems scheduling techniques. These techniques can be applied within the operating system, the file system, and within the network to provide resource guarantees to applications. We first start with a description of traditional real-time scheduling algorithms and then describe how these apply to multimedia systems.

6.1 Real-Time Systems Overview and Terminology

Real-Time systems are centered around the notion of *deadlines*, which represents the latest acceptable time for the presentation of a processing result. There are several types of deadlines. *Hard deadlines* are deadlines that *cannot* be missed. Missing a hard deadline typically results in catastrophic failure of the system. For example, airplane navigation systems must not report results too late. *Soft deadlines* are deadlines that, if missed, do not lead to catastrophic failure. Soft deadlines, however, imply that (i) not too many deadlines are missed and (ii) that they are not missed by much. Finally, *non-real-time deadlines* are those requests that have no deadlines.

In their original description of real-time systems, Liu and Leyland describe a number of fundamental assumptions regarding the tasks that need to be scheduled. These assumptions make their descriptions of real-time scheduling algorithms tractable. Each process that requires processing has a *deadline* for each request. The assumptions built around these deadlines include:

- Each task is periodic in nature and has a deadline at the end of every period
- Tasks require a fixed amount of processing per period and needs to complete the processing before moving to the next period
- The tasks are all independent of each other and that there are no dependencies in their processing requirements.
- Each task has a constant amount of work needed to be accomplished per period
- Any non-periodic tasks are special

Each real-time system consists of a number of tasks, or what is referred to as a *task set*. The task set is characterized by the following variables:

- $\tau_1, \tau_2, ..., \tau_m$ are the *m* periodic tasks that need to be scheduled
- $T_1, T_2, ..., T_m$ are the request periods of the *m* tasks
- $C_1, C_2, ..., C_m$ are the run times of each task (per request period)

For every period, each task has a *deadline* by which processing for the period needs to be completed. This deadline coincides with the request for the next period. We say that *overflow* occurs if at time *t*, *t* is the deadline of an unfulfilled request. That is, it missed the deadline.

Our goal then for real-time systems is to run all *m*tasks without any overflow. If a task set can be run without overflow in a real-time system, we refer to this as a *feasible task set*.

6.1.1 Admission Control

As we saw in the last chapter, one of the phases of resource management is determining whether a new task can be added to the system without over allocation our resource. This principle also holds for real-time systems. For real-time systems, this is commonly referred to as admission control. Suppose we have a set of *m* tasks that are feasible. If a new task is requested by a user, the real-time system needs to determine whether or not the new task along with the *m* other tasks is still feasible. If the *m+1* tasks are feasible, then the new task is *admitted* into the system. Otherwise, the new task is not allowed to run. Thus, the goal of the real-time system is to admit as many tasks as it can while still retaining a *feasible* task set.

6.1.2 Real-Time Schedulers

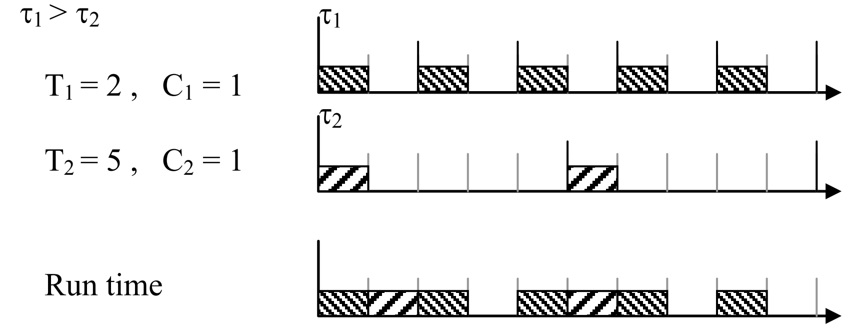
The primary mechanism to avoid overflow in real-time systems is the use of a scheduler and assigning priorities to tasks. There are two main types of schedulers for real-time systems. The first type uses static, fixed priorities. That is, the once a new task is admitted to the system, it is assigned a fixed priority for the duration it stays in the system. This class of schedulers is referred to as *rate monotonic schedulers*. The second type of real-time scheduler uses dynamic priorities for the tasks. As the tasks run, the real-time system adjusts the priorities to ensure that no overflow occurs. This type of scheduler is commonly referred to as deadline driven schedulers.

For the purposes of this boo, we assume that the scheduler is pre-emptive and priority driven. Thus, when a task *a* is running and a task *b* with higher priority exists and needs to run, task *a* is pre-empted and task *b* is then run.

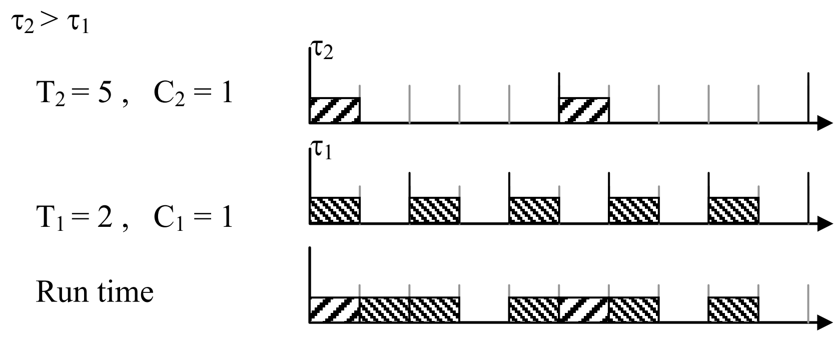
6.2 Rate Monotonic Schedulers

As previously mentioned, the priority of a task determines when the task is able to run. For rate monotonic scheduling (RMS), the tasks are given a fixed priority when they enter the system. Obviously, the priority assignment for a task has an impact on whether or not the task set if feasible or not.

As an example, consider the two tasks depicted below:



With the assignment of τ_1 having higher priority, we see that the task set if feasible. Furthermore, we also see that C_2 can also be increased to 2 without causing overflow. Now, let us consider the case where τ_2 has a higher priority than τ_1.



As shown in the above figure, running τ_2 with higher priority also results in a feasible task set. The big difference, however, is that C_2 cannot be increased any further. This is because *overflow* would occur at time 2 if C_2 were increased to 2. These two examples, give us insight on how to assign priorities with RMS.

In the first example, with priority τ_1 greater than priority τ_2, the following equation must be satisfied:

$$\left\lfloor \frac{T_2}{T_1} \right\rfloor C_1 + C_2 \le T_2 \qquad (1)$$

The first part of the equation determines how many times τ_1 fully runs (preempting τ_2). The second term adds in the amount of work τ_2 wants to accomplish. This needs to be less than the total amount of time available in T_2.

Using the same exact argument for the second case, where T_1 is less than T_2, we have the following:

$$C_1 + C_2 \le T_1 \qquad (2)$$

Multiplying (2) with $\left\lfloor \frac{T_2}{T_1} \right\rfloor$, we have the following:

$$\left\lfloor \frac{\mathrm{T}_2}{\mathrm{T}_1} \right\rfloor C_1 + \left\lfloor \frac{\mathrm{T}_2}{\mathrm{T}_1} \right\rfloor C_2 \le \left\lfloor \frac{\mathrm{T}_2}{\mathrm{T}_1} \right\rfloor T_1 \qquad (3)$$

The right side of the equation must always be less than T_2. Thus,

$$\left\lfloor \frac{\mathrm{T}_2}{\mathrm{T}_1} \right\rfloor C_1 + \left\lfloor \frac{\mathrm{T}_2}{\mathrm{T}_1} \right\rfloor C_2 \le \left\lfloor \frac{\mathrm{T}_2}{\mathrm{T}_1} \right\rfloor T_1 \le T_2 \qquad (4)$$

What this tells us is that if we have something that satisfies equation (2), then equation (1) is automatically satisfied. *HOWEVER*, if we have something that satisfies equation (1), this might not hold for equation (2).

As a result, this tells us that if we have two tasks τ_1 and τ_2 with $T_1 < T_2$ If priority(τ_2) > priority (τ_1) is feasible, then it is also feasible for priority(τ_1) > priority (τ_2). If priority(τ_1) > priority (τ_2) is *feasible*, then using the priority(τ_2) > priority (τ_1) is *not necessarily feasible.* Thus, for RMS, we should always schedule tasks with higher rates (smaller periods) with higher priorities.

Rate Monotonic Scheduling

RMS follows this principle, assigning higher rates to higher priorities. Furthermore, extending the argument above, it can be shown that *rate monotonic scheduling* is optimal in that no other assignment rule can schedule a tast set which cannot be scheduled by rate-monotonic priority assignment

To determine whether adding a task to a system is feasible or not using RMS, the following test is performed:

$$U_{proc} = \sum_{i=0}^{m} \frac{C_i}{T_i} \qquad (1)$$

Furthermore, using

$$U_{theoretical} = m(2^{1/m} - 1) \qquad (2)$$

If $U_{proc} < U_{theoretical}$, then the task set is automatically feasible. If the equation does not hold, then a more expensive test needs to be run. It is typically sufficient to just check whether or not $U_{proc} < 0.69$.

6.3 Earliest Deadline First Scheduling

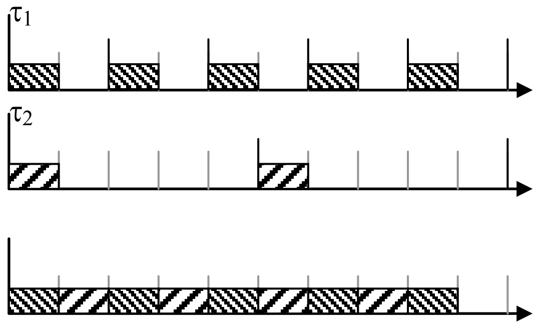
Rate monotonic scheduling, while using fixed priorities, results in extremely rigid execution of the tasks within the system. Consider the following:

$\tau_1 > \tau_2$

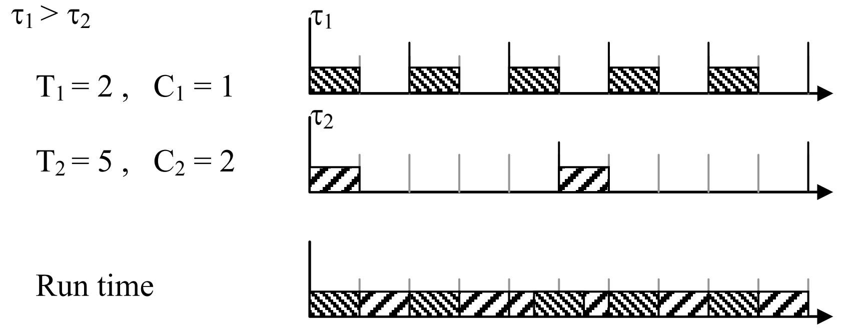
$T_1 = 2$, $C_1 = 1$

$T_2 = 5$, $C_2 = 2$

Run time



While the above task set is feasible, there is one extra unit of processing possibly available that τ_2 could spread over two periods. Of course, in RMS, increasing C_2 to 2.5 is not possible because in the third period, τ_1 interrupts τ_2. If τ_1 were able to delay by one half time unit, however, the task set becomes feasible. This case is shown below:



As shown above, moving the third running of τ_1 allows us to increase C_2 to 2.5 *without missing any deadlines*. The key to this however, is being able to change the priority mechanism in the third period to allow τ_2 to run.

This is, in fact, what deadline driven scheduling processing does. In the *Earliest Deadline First* scheduling algorithm, processes are assigned priorities according to the deadlines of all current requests. The main overhead in such a system is that the system needs to dynamically keep track of deadlines and adjust priorities accordingly.

Earliest Deadline First Scheduling

Earliest Deadline First (EDF) scheduling is a dynamic priority, pre-emptive scheduler that assigns priorities to tasks according to the deadline of current requests. It can be shown that using EDF scheduling results in the following theorem:

Theorem - When the deadline driven scheduling algorithm is used to schedule a set of tasks on a processor, there is no processor idle time prior to an overflow

This means that EDF scheduling can theoretically achieve 100% processor utilization (assuming 0 context switch and priority reordering overhead). Thus, the feasibility test of EDF is:

Theorem - For a give set of *m* tasks, the deadline driven scheduling algorithm is feasible if and only if

$$\sum_{i=0}^{m} \frac{C_i}{T_i} \leq 1.0 \qquad (1)$$

6.4 Real-Time Scheduling and Multimedia Systems

The RMS and EDF schedulers are designed for hard real-time systems. The techniques, however, are applicable to soft real-time systems such as multimedia processing. Obviously missing the presentation of a frame of video does not result in catastrophic failure. As the soft deadline implies, we do not expect to present a frame of video with a large amount of delay.

For the scheduling of multimedia data, there are a number of points that need to be considered. First, many multimedia applications have very similar periods. For example, most video is encoded at either 15, 24, or 30 frames per second. Second, multimedia data like video highly bursty in its resource requirements. Thus, the application does not have a fixed amount of processing it requires over time. Third, because multimedia data is somewhat scalable, it is more amenable to adapting its resource requirements. For example, if the application determines that the processor is unable to keep up, it can skip frames or skip some of the computations.

Between RMS and EDF schedulers, most implementations use EDF scheduling. The reason for this is that admission control is relatively easy to do. The processor asks for a percentage of the resource. Simply summing these up and checking against 1 is a very simple task.

6.5 Chapter Summary

In this chapter we have describe the basis for real-time scheduling systems. Real-time scheduling systems were originally designed for hard real-time systems where deadlines were not to be missed, ever! Multimedia systems, while requiring real-time scheduling over time, do have such hard deadlines. As a result, the techniques we learned in this chapter apply, but can be applied in a somewhat loose manner. For example, in scheduling bursty video, the process may require more processing to decompress a frame than it has been reserved. Presenting this result slightly late does not cause catastrophic failure.