Technical Report

CMU/SEI-90-TR-19
ESD-90-TR-220

An Analysis of Input/Output Paradigms for Real-Time Systems

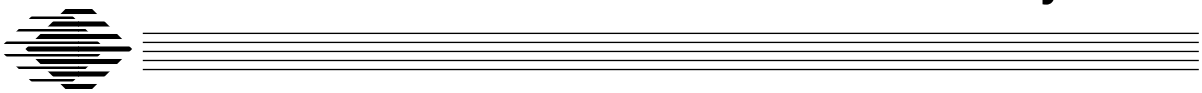
Mark H. Klein
Thomas Ralya

July 1990

Technical Report

CMU/SEI-90-TR-19
ESD-90-TR-220
July 1990

An Analysis of Input/Output Paradigms for Real-Time Systems



Mark H. Klein

Real-Time Scheduling in Ada Project

Thomas Ralya

IBM Federal Sector Division

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc. / 800 Vinial Street / Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page at http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

An Analysis of Input/Output Paradigms for Real-Time Systems

Abstract: The correctness of a real-time system with hard deadline requirements depends both on the logical correctness and on the timing correctness of the system. The principles of rate monotonic scheduling have proven to be very useful in providing a framework for designing, analyzing, and modifying the timing and concurrency aspects of real-time systems. This paper illustrates how to build a mathematical model of the schedulability of a real-time system, taking into consideration such factors as preemption, synchronization, non-preemptibility, interrupts, and process idle time. In particular, this paper illustrates how these principles can be applied to input/output interfaces (e.g., to devices or local area networks) to predict the timing behavior of various design alternatives.

1. Introduction

The primary characteristic that distinguishes real-time systems from non-real-time systems is the importance of time. The correctness of a real-time system depends not only upon its logical correctness but also its timing correctness [11, 15]. System complexity tends to compromise correctness unless there are techniques and methods for managing the complexity. Basic software engineering principles such as abstraction, encapsulation, and information hiding form the basis of methods and techniques that are used to manage logical complexity [13]. Rate monotonic scheduling theory offers a set of engineering principles for managing timing complexity [11].

A real-time program may be comprised of many processes (i.e., threads of execution) and the timing relationships between processes may be complex. The responsibility for implementing a set of processes may be held by one individual or, more likely, by many individuals possibly in different organizations. Further, the set of processes may change during the course of the development effort. Ultimately, the set of processes must be integrated to form a program that satisfies a set of real-time performance requirements.

A central focus of the Real-Time Scheduling in Ada (RTSIA) Project at the Software Engineering Institute (SEI) is to explore the use of rate monotonic scheduling theory for managing timing complexity and for understanding the timing behavior of realistic real-time problems. Input/output (I/O) processing plays an important role in real-time systems and, at the same time, poses several interesting problems for rate monotonic scheduling theory. The purpose of this report is to illustrate how to apply rate monotonic principles systematically to commonly used I/O paradigms.

1.1. An Analytical Framework

The notion of rate monotonic scheduling was first introduced by Liu and Layland in 1973 [5]. The term *rate monotonic* derives from a method of assigning priorities to a set of processes: assigning priorities as a monotonic function of the rate of a (periodic) process. Given this simple rule for assigning priorities, rate monotonic scheduling theory provides a simple inequality—comparing total processor utilization to a theoretically determined bound—that serves as a sufficient condition to ensure that all processes will complete their work by the end of their periods.

This fundamental theoretical result is the underpinning of a fairly comprehensive theory for analyzing the timing behavior and designing the concurrency structure of a real-time system. Liu and Layland's original result applied only to a set of non-interacting periodic processes. Subsequent work extended the applicability of rate monotonic scheduling to processes that synchronize to share data [10], to systems with aperiodic processing [4, 14], and to systems with mode change requirements [12]. As a result, the theory can be used to build a mathematical model that describes the ability of a system to meet its timing requirements. We refer to this as a *schedulability model*.

1.2. Considerations for Input/Output

One of the benefits of developing a schedulability model is that it requires a precise characterization of the execution timing behavior of a set of processes in terms of the parameters needed by the model. For example, to build a schedulability model that includes the effects of sharing data between processes, we must understand the circumstances under which lower priority processes can block higher priority processes by requiring exclusive access to the data. In order to build schedulability models for I/O paradigms we must precisely characterize input/output processing, explore relevant theoretical results, and then incrementally use the theory to understand how to model various aspects of different I/O paradigms.

In Chapter 2 we define a general model of processing that basically divides a process's work into three stages: input, processing, and output. We also define a classification of I/O devices. Two parameters are used to differentiate between device types: whether or not the device can handle more than one client process at a time and whether or not the device requires that the CPU participate in data movement. In Section 2.3 we develop notation that will be useful in performing analyses in the remainder of the paper. The theoretical results relevant to this paper are then summarized in Chapter 3.

Chapter 4 forms the heart of the paper. In this section we demonstrate how to apply the principles of rate monotonic scheduling theory to the general model of processing outlined earlier in the paper. First, we consider synchronous I/O paradigms. When synchronous I/O paradigms are used, control is not returned to the calling process until the operation is complete. Therefore, a process that uses synchronous I/O will perform one I/O operation at a

time; I/O operations do not overlap in time. On the other hand, asynchronous I/O operations may overlap since the calling process may start other work concurrent with the initiated I/O. We will explore the schedulability tradeoffs between these two paradigms by comparing their schedulability models. We will also explore other issues that impact the timing behavior of a set of processes including non-preemptible sections, interrupt processing, and process idle time (depending on the device, a process may be inactive during an I/O operation). Throughout Chapter 4 we develop a schedulability analysis of each situation that is presented.

2. Processing Model

The context of the discussion in this paper is real-time systems with hard deadlines. A *hard deadline* is a deadline that must be met; the software is considered to be malfunctioning if such a deadline is missed. We confine our discussion to uni-processor systems that employ logical concurrency. The term *process* will denote a unit of concurrency. We will further restrict our attention to periodic processes.[1] By *periodic* we mean that a process is initiated at regular intervals (periods) and has a deadline that is one period after it is initiated.

2.1. Input/Output Paradigms

We assume a general processing model that endlessly cycles through the following three stages as shown in Figure 2-1:

1. **Input:** Read data from one or more sources of input, which may be devices and/or data in main memory.

2. **Processing:** Compute output values, which are functions of all of the gathered input values.

3. **Output:** Write the results of the computations to one or more sinks, which may be devices and/or main memory.

The input and output resources (devices and/or memory storage) may be shared between processes in the system, and in that case will require mutually exclusive access.

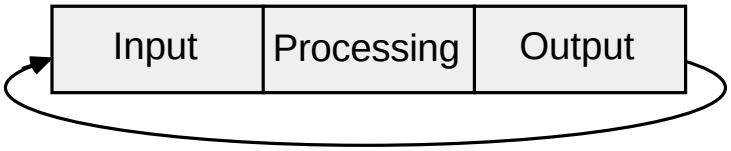


Figure 2-1: General Model for a Process

The input (output) stage of a process is simply a sequence of individual input (output) operations. We model an individual input (output) operation as occurring in three phases as illustrated in Figure 2-2.[2]

1. **Start I/O (St):** The time interval in which device interactions necessary to start an I/O operation are performed.

---

[1] It may seem overly restrictive to focus on periodic processing. However, much of the analysis is applicable to aperiodic processing. See [14] for a description of how to use the sporadic server algorithm to guarantee hard deadlines for aperiodic processes.

[2] Note that the ideas presented in this report are not limited to this particular model of processing. Arbitrary sequences of input operations, processing, and output operations can also be analyzed using the principles of rate monotonic scheduling theory.

2. **I/O Service (Srv):** The time interval in which the data is actually manipulated and/or moved.

3. **I/O Completion (Cpt):** The time interval which starts when the device signals that I/O has completed and ends when control is returned to the initiating process.

When considering shared data in main memory as the resource, the I/O service phase is the only relevant phase. In this case, this phase reflects the amount of time it takes to perform an operation on the shared data. When considering devices, the I/O service phase reflects the amount of time it takes to move data between main memory and another destination.

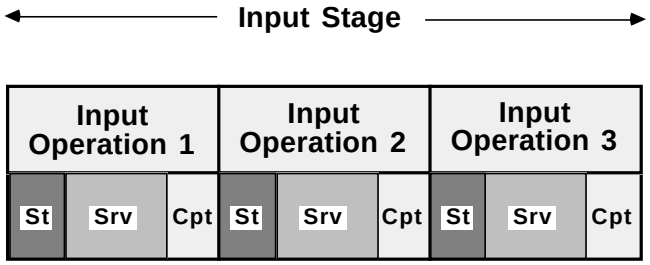


Figure 2-2: Input Stage in Detail

We will consider variations of two common I/O paradigms: synchronous and asynchronous. When a synchronous I/O operation is performed, control is returned to the calling process only after the entire I/O operation is complete. A process that employs synchronous I/O completes all phases of an I/O operation before it starts the next I/O operation. When asynchronous I/O is performed, control is returned to the calling process immediately after the operation is started, enabling the calling process to perform other work concurrent with the I/O. In particular, the asynchronous paradigms perform the Start-IO phase of several distinct I/O operations, allowing the I/O-service phase of several I/O operations to proceed concurrently. The characteristics of the I/O device and the software interface to the device are factors to be considered when choosing between synchronous and asynchronous paradigms.

We frequently refer to a process that makes I/O requests as the *client* process. We will assume that I/O capabilities are provided to the client via a software interface. The syntactic details of the I/O interface are not of concern to us; instead, we are interested in the semantics of the interaction between the client and the device. The following issues have an impact upon the analysis of various paradigms:

- **Non-preemptible sections**. Is any portion of the I/O operation non-preemptible?

- **Non-interruptible sections**. Are interrupts disabled for any portion of the I/O operation?

- **Idle time.** Is the client process inactive[3] for any portion of the I/O operation?

- **Mutual exclusion.** Does the I/O operation provide mutually exclusive access to the device?

- **Interrupts.** Is the device operating in an interrupt-driven mode or in a polling mode?

The properties of the I/O service will have an impact on the client's ability to satisfy its timing constraints and may impact other processes as well. Characteristics of the device naturally determine the nature of the client's interaction with the device. Thus, it is important to understand certain aspects of a device's behavior in order to understand the timing behavior of an I/O process.

2.2. Models of Device Interactions

To be clear about the assumptions that we are making concerning device behavior, several classes of devices are described below.

- **CPU Dependent.** This class of device requires the CPU to be active in moving the data. The Motorola Z8530 [6] serial interface, commonly used on Motorola single board computers, falls into this class.

- **Single Request.** This class of device does not require the CPU to be active in moving the data. The I/O device operates physically concurrent with the CPU. Devices in this class only support one outstanding I/O request at a time. Most direct memory access (DMA) controllers fall into this class.

- **Multiple Request.** This class of device does not require the CPU for data movement and also operates physically concurrent with the CPU. However, devices in this class support multiple outstanding I/O requests. Some local area network adapters fit into this class.

Performing an I/O operation using a device in any of the above classes involves the use of many resources. Cognizance of and planning for resource contention is important in determining whether or not a process will be able to meet its deadline. For example, before a process can perform an I/O operation it must acquire the CPU. The process may then need to acquire several I/O buffers. If memory is a scarce resource, this may cause the process to wait. The I/O device itself may be a shared resource. If the device is being used by a lower priority process, a higher priority process may be delayed. A common backplane bus may be used to facilitate communication between the CPU and devices. Bus arbitration protocols may have an impact on a process's ability to meet its deadline. For example, bus

[3] An idling (or inactive) process in this context is waiting for an I/O service to be completed. Lower priority tasks have an opportunity to execute when the client is inactive.

cycles may be lost to DMA devices in the presence of I/O activity. This effectively reduces the number of cycles available to a process that also needs access to the bus and consequently introduces delays in the process's I/O stages. (See [8] for a comprehensive discussion of "cycle stealing.") For multiple-request devices, scheduling of requests within the device itself is also an issue. Additionally, multiple-request devices are likely to have a limit for the number of outstanding I/O requests. Another factor when considering the timing behavior of processes performing I/O is the interrupt control logic of the processor. In this paper we focus on issues concerning the use of devices and memory-resident data by one or more processes. We assume that there is no contention for the other resources mentioned above (i.e., I/O buffers are readily available, cycle stealing is negligible, and multi-request devices support a large number of outstanding requests).

Section 2.3 introduces some notation and terminology.

2.3. Notation and Terminology

In general, we assume that there are *n* processes on a uni-processor. One or more of these processes may be a process that performs I/O as described in the previous section. Any given process *i* will be denoted by τ_i.

The term *schedulability* means the ability of a process or a set of processes to meet deadlines. We explore later how various characteristics of a client process's interactions with different types of resources affect its schedulability.

There are several parameters of a process that we refer to many times in later sections. C_i and T_i represent the execution time and period, respectively, associated with process τ_i. Assume that the numbering of the processes is such that the following relationship holds:

$$T_1 \leq T_2 \leq \cdots \leq T_n$$

The *CPU utilization* of process τ_i is the ratio of a process's execution time to its period. The CPU utilization of a set of processes is the sum of the utilizations of the individual processes.

$$CPU\ Utilization\ of\ a\ Set\ of\ Processes = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n}$$

Let $Res(\tau_i)$ be the set of resources that process τ_i uses (which includes both devices and shared data) and let $Dev(\tau_i)$ be the subset of those resources that are devices. The following expression summarizes this relationship:

$$Dev(\tau_i) \subseteq Res(\tau_i)$$

Devices may be used in the input and/or output stages. Therefore, let the set of devices that are used for input and output by process τ_i be denoted by $InpDev(\tau_i)$ and $OutDev(\tau_i)$

respectively.[4] The following expression summarizes this relationship:

$$InpDev(\tau_i) \cup OutDev(\tau_i) = Dev(\tau_i)$$

As previously stated, the execution time of process τ_i is represented by C_i. There are also subcomponents of execution time that are of interest. Let $r \in Res(\tau_i)$ denote a resource that is used by τ_i. The amount of time that process τ_i spends performing an I/O operation with resource r is $C_{i,r}$. Additionally, $C_{i,p}$ is the amount of time that τ_i spends in its processing stage. Therefore, we have the following relationship:[5]

$$C_i = C_{i,p} + \sum_{r \in Res(\tau_i)} C_{i,r}$$

This expression states that the total execution time for τ_i is the sum of the execution times associated with performing I/O with all resources used by τ_i, plus the execution time associated with the τ_i's processing stage. Note that process τ_i may use the same resource more than once per period and may use it for input and output. If it is necessary to distinguish between multiple uses of the same resource, we will let $C_{i,r,k}$ denote the *k*'th use of resource *r* by process τ_i during any given period. Otherwise, the third subscript will be omitted.

In the previous section, we subdivided a client process's interaction with a device into phases: start I/O, I/O service, and I/O completion. Let $d \in Dev(\tau_i)$ denote a device that is used by τ_i.

- $St(C_{i,d,k})$ denotes the amount of time process τ_i spends in the start I/O phase of an individual I/O operation using device d.

- $Srv(C_{i,d,k})$ denotes the amount of time process τ_i spends in the I/O service phase of an individual I/O operation using device d.

- $Cpt(C_{i,d,k})$ denotes the amount of time process τ_i spends in the I/O completion phase of an individual I/O operation using device d.

When certain paradigms are used, the client process will not actually be executing during the service time phase; execution will be suspended and the client will be inactive while the device is performing I/O. We will refer to this time as *idle* or *inactive time*. If a process is inactive, it will be inactive during the I/O service phase of I/O operations. Therefore, if an individual I/O operation is not inactive, the execution time associated with the operation is:

$$C_{i,d,k} = St(C_{i,d,k}) + Srv(C_{i,d,k}) + Cpt(C_{i,d,k})$$

---

[4] Note that an individual device may be used for both input and output. In this case the device would be a member of both $InpDev(\tau_i)$ and $OutDev(\tau_i)$.

[5] We will assume that the amount of computation that a process performs between individual I/O operations is negligible.

On the other hand, if an individual I/O operation is inactive during its I/O service phase then:

$$C_{i,d,k} = St(C_{i,d,k}) + Cpt(C_{i,d,k})$$

Finally, let *LowRes(τ_i)* denote the set of resources that are used by processes with priorities less than τ_i's priority and *HiRes(τ_i)* denote the set of resources used by processes that have a priority which is greater than or equal to τ_i's priority.

3. Review of Rate Monotonic Theory

The analysis of the schedulability of various I/O paradigms will be performed by using a theory of real-time systems which is based on rate monotonic scheduling theory. Rate monotonic scheduling theory provides analytical mechanisms for understanding and predicting the execution timing behavior of real-time systems. The basic theory, introduced in a seminal paper written by Liu and Layland [5], gives us a rule for assigning priorities to periodic processes and a formula for determining if a set of periodic processes will meet all of their deadlines. A large body of work resulting from the Advanced Real-Time Technology[6] and the Real-Time Scheduling in Ada[7] Projects at Carnegie Mellon University extends this basic result so that the theory addresses process synchronization, aperiodic processing, mode change, and other practical issues that contribute to the complexity of the timing behavior of real-time systems [3, 4, 10, 14]. This section reviews some of the relevant results that are used later in the paper.

3.1. Basic Results of Rate Monotonic Scheduling

Our examination of the schedulability of various I/O paradigms addresses the following two questions about a process that performs I/O:

1. How do other processes affect the schedulability of the process?

2. How does the process affect the schedulability of other processes?

These questions serve as a starting point to introduce rate monotonic theory.

First, note that we assume a priority-based preemptive scheduling discipline.[8] Initially, consider a set of independent periodic processes, where *independent* means that the processes do not have synchronization requirements and *periodic* means the processes are initiated at regular periods and have deadlines at the end of the period. Under these assumptions, only higher priority processes can affect the schedulability of a particular process. Higher priority processes delay a process's completion time by preempting it. This is reflected in the following theorem [5].

> **Theorem 1:** The rate monotonic algorithm assumes priority-based preemptive scheduling, where a process's priority is based on its period; processes with shorter periods (i.e., higher frequencies) are assigned higher priorities. A set of *n* independent periodic processes scheduled by the rate monotonic algorithm will always meet their deadlines, for all task phasings, if

[6] A project in Carnegie Mellon University's School of Computer Science.

[7] A project in Carnegie Mellon University's Software Engineering Institute.

[8] Rate monotonic principles have been used to analyze non-preemptive scheduling disciplines as well.

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \leq U(n) = n(2^{1/n}-1)$$

Basically, if the utilization of the process set is less than a theoretically determined bound, then the set of processes is guaranteed to meet all of its deadlines.

Corollary 2: Given a set of *n* independent periodic processes scheduled by the rate monotonic algorithm, a particular process, τ_k, $k \leq n$, will always meet its deadline if:

$$\frac{C_1}{T_1} + \cdots + \frac{C_k}{T_k} \leq U(k) = k(2^{1/k}-1)$$

From this result we can see that the only factors that determine the schedulability of process τ_k are the utilization of higher priority tasks and the utilization of the process τ_k itself.

As indicated above, there is a set of assumptions that are prerequisites for this result (see [1]):

- Process switching is instantaneous.
- Processes account for all execution time (i.e., the operating system does not usurp the CPU to perform functions such as time management, memory management, or I/O).
- Process interactions are not allowed.
- Processes become ready to execute precisely at the beginning of their periods.
- Process deadlines are always the start of the next period.
- Processes with shorter periods are assigned higher priorities; the criticality of processes is not considered.

The following set of results allows us to relax these assumptions and thus apply the scheduling theory to a wide class of realistic real-time problems, such as the analysis of various I/O paradigms.

Corollary 3: Let worst-case context switching time between processes be denoted by C_s. Also, define $C_i' = C_i + 2C_s$. A set of *n* independent periodic processes with worst-case context switching time of C_s that is scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if:

$$\frac{C_1'}{T_1} + \cdots + \frac{C_n'}{T_n} \leq n(2^{1/n}-1)$$

The execution time of process τ_i is effectively being inflated to include context switching overhead. As described in [1], when a process preempts a lower priority process, the ex-

ecution state of the lower priority process is saved and the execution state of the higher priority process is established. When the higher priority process completes its processing and relinquishes the CPU to a lower priority process, its execution state is saved and the state of the lower priority process is reestablished. The context switches for (1) preemption of the lower priority process and subsequent (2) resumption of its execution account for the $2C_s$ added to the execution time of the preempting process.

The discussion up to this point assumes that a process's execution is always consistent with its rate monotonic priority. Consider the following example:

> **Example 1:** Two processes have been assigned rate monotonic priorities with process τ_1 the highest priority. Process τ_2 starts to execute and calls a system service, a portion of which involves a non-preemptible section of code. Immediately after this call, τ_1 becomes ready to execute but cannot preempt τ_2 while it is in this non-preemptible section. Thus, the higher priority process has to wait until the system service completes before it can preempt the lower priority process.

This example illustrates one way in which a process that has been assigned a higher rate-monotonic priority can be delayed by a lower priority process. This delay time is known as *priority inversion* or *blocking*. Interrupts represent another potential source of blocking. The following result generalizes the previous results to include the effects of blocking.

> **Corollary 4:** Given a set of *n* independent periodic processes scheduled by the rate monotonic algorithm, let B_k be the worst-case total amount of blocking that process τ_k can incur during any period. Process τ_k will always meet its deadline if:
>
> $$\frac{C_1'}{T_1} + \cdots + \frac{C_k'}{T_k} + \frac{B_k}{T_k} \leq k(2^{1/k} - 1)$$

The following lemma is a generalization of the above corollary.

> **Lemma 5:** Given a set of *n* independent periodic processes scheduled by the rate monotonic algorithm, let B_i be the worst-case total amount of blocking that process τ_i can incur during any period. The set of processes will meet all deadlines for all phasings if:
>
> $$\frac{C_1'}{T_1} + \frac{B_1}{T_1} \leq 1(2^{1/1} - 1) \quad \textit{and}$$
> $$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{B_2}{T_2} \leq 2(2^{1/2} - 1) \quad \textit{and}$$
> $$\cdots$$
> $$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \cdots + \frac{C_k'}{T_k} + \frac{B_k}{T_k} + \leq k(2^{1/k} - 1) \quad \textit{and}$$
> $$\cdots$$
> $$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \cdots + \frac{C_n'}{T_n} + \leq n(2^{1/n} - 1)$$

The inequalities explicitly show how blocking affects the schedulability of a set of processes and why it is desirable to minimize blocking.

Process synchronization is another common source of blocking. When more than one process requires mutually exclusive access to a resource, processes must synchronize. If a lower priority process has locked a resource and is then preempted by a higher priority process which executes until it needs to access the resource but is then forced to wait, the higher priority process is blocked. The priority ceiling protocol (PCP), first described in [10], is one of a class of inheritance protocols; PCP reduces the effects of blocking and prevents mutual deadlock.

The PCP employs two concepts: priority inheritance and the priority ceiling. When a high priority process is waiting for a lower priority process to relinquish access to a shared resource, priority inheritance comes into play. Priority inheritance prohibits a medium priority process from prolonging the actual amount of time a resource is locked by a lower priority process. Without priority inheritance, a medium priority process can preempt the lower priority critical section and prolong the period of blocking. To prevent this, *priority inheritance* allows the lower priority process to inherit the blocked process's higher priority for the duration of the critical section. Thus, priority inheritance prevents the medium priority process from preempting the critical section, which is now executing at a high priority. The basic priority inheritance protocol is described in [10]. Priority inheritance leads to the following result [10]:

> **Theorem 6:** Under the basic priority inheritance protocol, if a process shares **m** resources with lower priority processes, then it can be blocked at most **m** times per period due to process synchronization (provided that the process does not become inactive).

It is not hard to imagine that a high priority process requires data from several resources that are all locked at the time it preempts and tries to acquire the data. The low priority process locks a resource, is then preempted by a slightly higher priority process that locks another resource, and so on. The high priority process will execute until it needs data from the first resource and then it will be blocked. The blocking process will inherit the blocked process's priority and, after its critical section, relinquish the resource. The high priority process will use the resource and then be forced to wait for access to the second resource that it needs and so on. The PCP reduces this blocking time [10].

> **Theorem 7:** Under the priority ceiling protocol, a process which shares resources with lower priority processes can be blocked only once per period (provided it does not become inactive when it is not accessing a resource) for the duration of a single critical section.

One can get an intuitive understanding of this property by examining the sources of blocking for any process, τ_i. Process τ_i can be blocked by any lower priority process with which it shares a resource (this is referred to as *direct blocking*). It can also be blocked by any lower priority process that shares a resource with a higher priority process. The lower priority process can inherit a higher priority when it is blocking a higher priority process, thereby

delaying process τ_i (this is known as *push-through blocking*). We will call the set of processes that can block process τ_i its *blocking set*. The PCP allows only one process in τ_i's blocking set to be locking resources at any given time. Therefore, when τ_i preempts a lower priority process it can only be blocked once due to process synchronization. The concept of a priority ceiling is used to accomplish this.

Associated with every semaphore or monitor that protects a shared resource is an attribute known as the priority ceiling. The *priority ceiling* is the highest priority at which a critical section associated with the resource can be executed, which is also the priority of the highest priority process that uses the resource. The priority ceiling rule of the priority ceiling protocol prohibits a process from locking a resource unless the process's priority is strictly greater than the priority ceiling of all semaphores locked by other processes. The blocking set of any process is the set of processes that use semaphores (or monitors) that have a priority ceiling greater than or equal to the process's priority. Effectively, the priority ceiling rule allows only one process in the blocking set to have locks at any given time.

> **Lemma 8:** One can emulate the priority ceiling protocol by ensuring that critical sections are executed at the ceiling priority.

If a critical section is executed (without becoming inactive) at the priority of the priority ceiling of the protected resource, then no other processes in the blocking set will be permitted to preempt the critical section. This effectively emulates the priority ceiling rule.

Another phenomenon that affects the schedulability of a process is idle time. Clearly when a process becomes inactive this has a direct impact on the schedulability of this process (see Figure 3-1). Another term is needed in the scheduling inequality for this process to account for the idle time. A much less obvious effect is that idle time can also reduce the schedulability of lower priority processes. This is known as the *deferred execution effect*, since execution is deferred for the duration of the idle time. This is discussed in [9, 4]. When a higher priority process's execution is deferred, there is a window of time where a lower priority process experiences more preemption than is normally permitted under rate monotonic scheduling. One can imagine that all of the higher priority process's execution is deferred so that the process completes its execution at the end of its period and then immediately resumes execution at the beginning of its next period (see Figure 3-2).

> **Lemma 9:** The deferred execution effect caused by a higher priority process can be accounted for by adding a blocking term to the inequalities of lower priority processes. This term is the minimum between the duration of idle time and the amount of execution time that has been deferred [9].

Consider, for example, Figure 3-2(a). Without idle time, process τ_2 has 5 units of execution time available that it can use without missing a deadline. In this case, the minimum between the duration of idle time (4 units) and the amount of execution time that was deferred (1 unit) is 1 unit. Figure 3-2(a) illustrates that process τ_2 has only 4 units of available execution time (a schedulability penalty of 1 unit) when the higher priority process idles. Figure 3-2(b) also illustrates a deferred execution penalty. In this case, the penalty is equal to the duration of execution, whereas in Figure 3-2(a) the penalty is equal to the amount of execution time that is deferred.

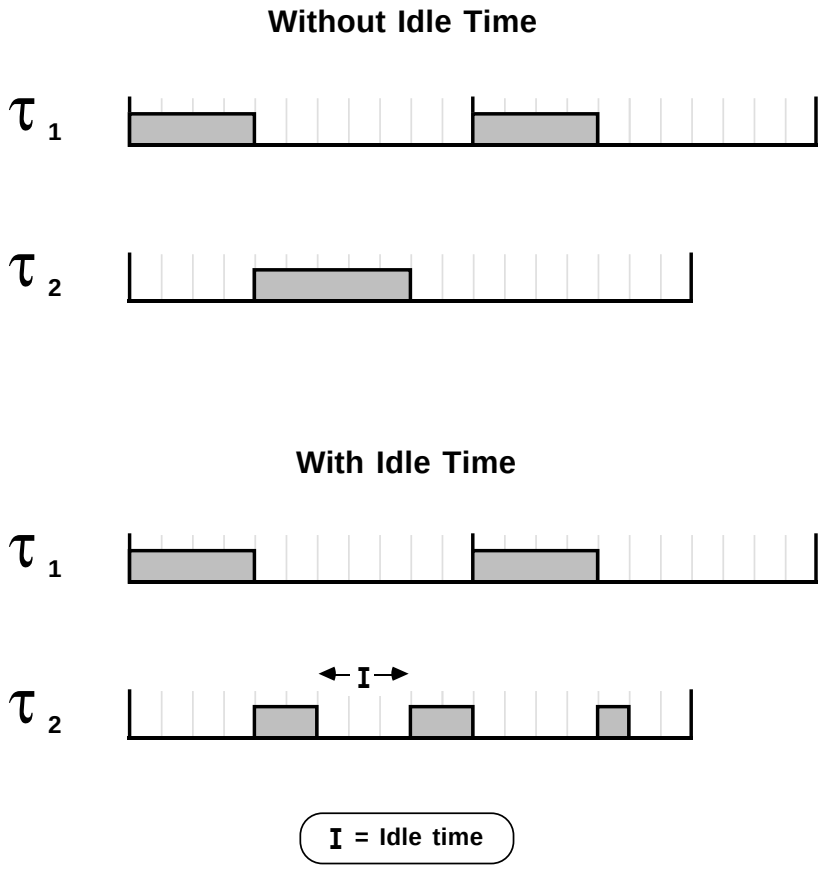


Figure 3-1: Effects of Idle Time

3.2. Schedulability Models

The following set of inequalities can be thought of as a mathematical model of the timing behavior of a set of **n** periodic processes.

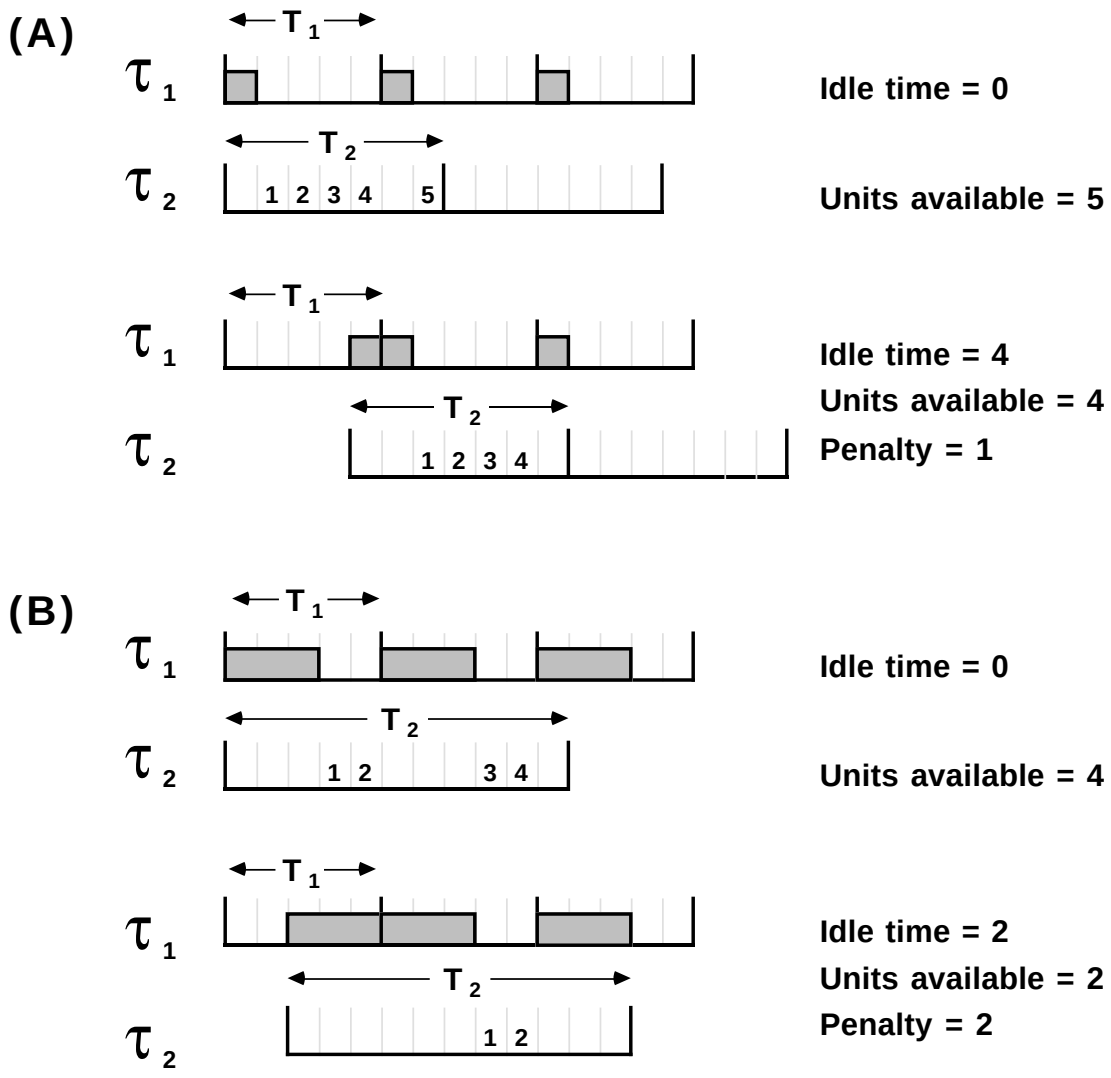


Figure 3-2: Deferred Execution Effect

$$\frac{C_1'}{T_1} + \frac{X_1}{T_1} \leq 1(2^{1/1} - 1) \ \textit{and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{X_2}{T_2} \leq 2(2^{1/2} - 1) \ \textit{and}$$

$$\cdots$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \cdots + \frac{C_k'}{T_k} + \frac{X_k}{T_k} + \ \leq k(2^{1/k} - 1) \ \textit{and}$$

$$\cdots$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \cdots + \frac{C_n'}{T_n} + \frac{X_n}{T_n} + \ \leq n(2^{1/n} - 1)$$

X_i is a term that contains all of the process-specific effects for process τ_i, which include blocking time (due to synchronization, interrupts, and other sources), idle time, and the deferred execution penalty. It is a model in the sense that it predicts the schedulability of the set of processes given a set of parameters, namely execution times, periods, and process-specific effects. Building a schedulability model for a set of processes necessitates understanding how to address the two questions at the beginning of Section 3.1 for each process, which allows one to build the set of inequalities one process at a time.

3.3. Example Problem

In order to illustrate the application of rate monotonic theory to several I/O paradigms, we use an example set of five processes. A data-flow diagram is shown in Figure 3-3. Devices are denoted as d_i and data stores as s_i. Table 3-1 shows the resources that are used by each process.[9] The example as a whole involves five different devices and four different data stores. Recall that we assume that the numbering of the processes is such that τ_1 has the shortest period and consequently has been assigned the highest priority. In general, the priority of process τ_i is higher than the priority of process τ_{i+1}. Assume that the priority ceiling protocol is in effect unless otherwise stated.

1. **Process τ_1** does not use any resources. Even though it is an independent periodic process, there are circumstances under which its ability to meet its deadline is affected by lower priority processes.

2. **Process τ_2** gathers data first from a device (d_1) and then from a data store (s_1), processes the data, and then writes the results to s_1.

[9] Since it will be useful to be able to look at the figure and the table while reading the examples later in the paper, the table and the figure have been duplicated in Appendix B.

3. **Process τ_3** gathers data from the three resources: s_1, d_2, and then d_3. It then performs calculations on the data and writes results to s_2 and sends output to device d_4. Note that this process shares data stores with processes τ_2 and τ_5 and shares a device with process τ_4.

4. **Process τ_4** gathers data from two data stores that are not shared with any other processes in this example and writes to device d_4, which it shares with τ_3.

5. **Process τ_5** gathers data from two data stores that are shared with higher priority processes and sends output to device d_5, which is dedicated to this process.

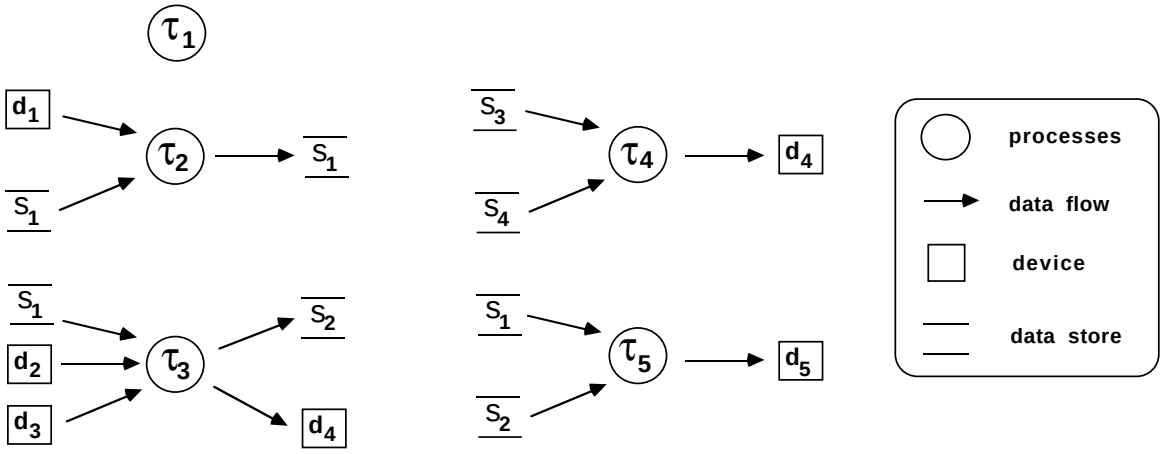


Figure 3-3: Process/Resource Relationships in the Example Problem

	d_1	d_2	d_3	d_4	d_5	s_1	s_2	s_3	s_4
τ_1									
τ_2	x					x			
τ_3		x	x	x		x	x		
τ_4				x				x	x
τ_5					x	x	x		

Table 3-1: Process/Resource Relationships in the Example Problem

4. Input/Output Paradigms

This chapter shows how to apply the theoretical results of the previous section to a set of processes, a subset of which perform I/O. Additionally, we hope to illustrate how the theory can be used to elucidate the tradeoffs between using various I/O paradigms. Basically, we will present variations of synchronous and asynchronous I/O paradigms.

When considering the variety of cases presented in this section, we always focus on a single period of a single I/O process, τ_k. We then strive to answer two fundamental questions:

1. How do other processes affect the schedulability of the performing I/O process τ_k?

2. How does the performing I/O process τ_k affect the schedulability of other processes?

In effect, answering these two questions is like specifying a schedulability interface for process τ_k: importing the information needed to determine its schedulability and exporting the information needed to determine the schedulability of other processes. This approach facilitates a separation of concerns, allowing us to focus our attention on a single process as we vary different aspects of its execution.

4.1. Synchronous I/O

4.1.1. Preemptible Service

In this first case the client process (i.e., the process making I/O requests) employs synchronous I/O (i.e., the client process waits for completion of the I/O operation). Moreover, the client process does not experience idle time (i.e., lower priority processes are not given an opportunity to execute) and the process is completely preemptible. Each resource is locked for the entire duration of the I/O request. Figure 4-1 illustrates an implementation paradigm[10] for this type of I/O service using Ada pseudo-code.

[10] By implementation paradigm we mean a specification for the characteristics of an implementation but not the implementation per se.

```
package IO_Services is
   procedure Read( <Buffer> );
   procedure Write( <Buffer> );
end IO_Services;


package body IO_Services is
   procedure Read( <Buffer> ) is
   begin
      IO_Monitor.Read( <Buffer> );
   end Read;

   procedure Write( <Buffer> ) is
   begin
      IO_Monitor.Write( <Buffer> );
   end Write;

end IO_Services;


task body IO_Monitor is
begin
  loop
      select
        accept Read( <Buffer> )  do
            Start IO;
            Poll device for I/O Completion;
            I/O Completion;
        end Read;
      or
        accept Write( <Buffer> )  do
            Start IO;
            Poll device for I/O Completion;
            I/O Completion;
        end Write;
      end select;
  end loop;
end IO_Monitor;
```

Figure 4-1: Synchronous Service with No Idle Time

One situation where this model applies is when the CPU polls the device to determine when the device has completed an I/O request (i.e., completed its I/O service phase). This is illustrated in Figure 4-1. The three phases are explicitly shown for the `Read and Write` operations. Requisite buffer manipulation and device control are implicit in the Start-IO and I/O-completion phases of the `Read and Write` operations. This example assumes a single-request device where both I/O operations poll to determine when the device has finished moving data from an external source to processor memory or vice versa. This I/O paradigm is also applicable in the case where the device is CPU dependent (i.e., the CPU is involved in the movement of data), as described in Section 2.2. Under these circumstances, a request to acquire data from a device has the same schedulability properties as a request to read/write memory-resident shared data. In the previous section we explained that the following generic inequality is used to model the schedulability of a particular process:

$$\frac{C_1'}{T_1} + \cdots + \frac{C_{k-1}'}{T_{k-1}} + \frac{C_k'}{T_k} + \frac{X_k}{T_k} \leq k(2^{1/k} - 1)$$

Assuming that the priority ceiling protocol has been implemented or is being emulated for all processes, then

$$X_k = B_k = max\left(C_{j,r} \mid j = k+1, \ldots, n;\ r \in DB(\tau_k) \cup PTB(\tau_k)\right)$$

where

$$DB(\tau_k) = Res(\tau_k) \cap LowRes(\tau_k)$$

$$PTB(\tau_k) = \{\ r \mid r \notin Res(\tau_k) \wedge r \in HiRes(\tau_k) \wedge r \in LowRes(\tau_k)\ \}$$

This means that the process-specific (X_k) term in the inequality is solely comprised of blocking time. Blocking time may be direct blocking and/or push-through blocking (see page 15). The set $DB(\tau_k)$ is the set of resources that may cause direct blocking and $PTB(\tau_k)$ is the set of resources that may cause push-through blocking.

Example 2: Referring to the example problem introduced in Section 3.3, we will focus on process τ_3. Recall that the general set of inequalities that models this set of processes is:

$$\frac{C_1'}{T_1} + \frac{X_1}{T_1} \leq 1(2^{1/1} - 1) \ \ and$$
$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{X_2}{T_2} \leq 2(2^{1/2} - 1) \ \ and$$
$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3'}{T_3} + \frac{X_3}{T_3} \leq 3(2^{1/3} - 1) \ \ and$$
$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3'}{T_3} + \frac{C_4'}{T_4} + \frac{X_4}{T_4} \leq 4(2^{1/4} - 1) \ \ and$$
$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3'}{T_3} + \frac{C_4'}{T_4} + \frac{C_5'}{T_5} + \frac{X_5}{T_5} \leq 5(2^{1/5} - 1)$$

In this example, processes are affected only by preemption and blocking due to shared resources. Process τ_3 shares resources with both lower and higher priority processes.

$$Res(\tau_3) \cap LowRes(\tau_3) = \{d_4, s_1, s_2\}$$
$$Res(\tau_3) \cap HiRes(\tau_3) = \{s_1\}$$

Since we are assuming that the PCP is in effect, τ_3 can be blocked for at most the duration of a single critical section of a lower priority process. Therefore, the blocking incurred by τ_3 is:

$$B_3 = max(\ C_{4,d4}, C_{5,s1}, C_{5,s2}\ )$$

The contribution of τ_3 to the blocking of higher priority processes is $C_{3,s1}$; process τ_3's contribution must be combined with other sources of blocking. The entire set of blocking terms for this example is:

$$X_1 = B_1 = 0$$
$$X_2 = B_2 = max(\ C_{3,s1}, C_{5,s1}\ )$$
$$X_3 = B_3 = max(\ C_{4,d4}, C_{5,s1}, C_{5,s2},\ )$$
$$X_4 = B_4 = max(\ C_{5,s1}, C_{5,s2})$$
$$X_5 = B_5 = 0$$

Notice that the source of blocking for process τ_4 is push-through blocking.

4.1.2. Considerations for Non-Preemptibility

Non-preemptible sections can result in blocking. Consider the case where I/O service is not only performed in a mutually exclusive manner, but is also non-preemptible. Perhaps the service is non-preemptible because of device requirements or merely because the I/O service was implemented in this manner. All other assumptions remain the same. As we illustrated in Section 3.1, non-preemptible sections represent a source of blocking to higher priority tasks. The following example illustrates the analysis for this case.

Example 3: Assume that I/O service for all devices is non-preemptible. Once again, we first consider τ_3.

In addition to the blocking term in the previous example there is another source of blocking for τ_3; τ_5 accesses d_5 in a non-preemptible section. Non-preemptibility is similar, in effect, to PCP. When a client is accessing a resource in a non-preemptible section, higher priority processes are prevented from executing and thus are prevented from locking other resources. The same effect would be achieved if the priority ceiling associated with the resource was set to be the highest priority in the system (independent of the clients that use the resource).[11] Of course, this causes blocking not caused by PCP; however, PCP's "blocked at most once" property is preserved. Therefore, the blocking term for process τ_3 is:

$$B_3 = max\Big(max(C_{4,d4}, C_{5,s1}, C_{5,s2}), C_{5,d5})\Big)$$

Since τ_3 accesses devices d_2, d_3, and d_4 in a non-preemptible manner, it becomes a source of blocking to higher priority processes. Its contribution to blocking resulting from non-preemptibility is $max(C_{3,d2}, C_{3,d3}, C_{3,d4})$. The entire set of blocking terms for this example becomes:

$$B_1 = max(C_{2,d1}, C_{3,d2}, C_{3,d3}, C_{3,d4}, C_{4,d4}, C_{5,d5})$$
$$B_2 = max\Big(max(C_{3,s1}, C_{5,s1}), C_{3,d2}, C_{3,d3}, C_{3,d4}, C_{4,d4}, C_{5,d5} \Big)$$
$$B_3 = max\Big(max(C_{4,d4}, C_{5,s1}, C_{5,s2}), C_{5,d5})\Big)$$
$$B_4 = max\Big(max(C_{5,s1}, C_{5,s2}), C_{5,d5}\Big)$$
$$B_5 = 0$$

Notice that in two of the blocking terms nested max functions were used. This is to emphasize the different sources of blocking and the composition of those different sources of blocking.

4.1.3. Considerations for Idle Time

The single-request and multiple-request devices (described in Section 2.2) allow for physical concurrency between the CPU and the device. This allows the client to relinquish the CPU to lower priority processes while awaiting I/O completion. Recall that this period of time when the client is not executing is referred to as *idle time* or *inactive time*. This section addresses the schedulability ramifications of process idle time.

Assume that I/O completion is signalled by a device interrupt which terminates the period of client inactivity and eventually results in control being returned to the client. Additionally, assume that the CPU is non-preemptible from the time the interrupt occurs until control is returned to the client process (i.e., the client is non-preemptible during the I/O completion phase of the I/O request). Figure 4-2 illustrates an implementation paradigm for the IO_Monitor for this case. Notice that this monitor waits for an interrupt to signal the comple-

[11] Actually, the effect is identical to using PCP emulation but setting the server's priority to be higher than any clients in the system. (Recall, when PCP emulation is used the critical section is executed at a priority which is equal to the priority ceiling of the semaphore.)

tion of the I/O service phase, whereas the monitor illustrated in Figure 4-1 polls the device. Now consider the schedulability characteristics of this type of interaction with a device.

```
task body IO_Monitor is
begin
  loop
      select
        accept Read( <Buffer> )  do
            Start IO;
            Wait for I/O Interrupt;
            I/O Completion;
        end Read;
      or
        accept Write( <Buffer> )  do
            Start IO;
            Wait for I/O Interrupt;
            I/O Completion;
        end Write;
      end select;
  end loop;
end IO_Monitor;
```

Figure 4-2: Synchronous Service with Idle Time

Idle time will have a direct impact on the client process's ability to meet its deadline. The idle time must be accounted for in the process's inequality in the same manner as blocking. Additionally, each time the client process becomes idle and then resumes execution, it incurs two additional context switches ($2C_s$), which must be accounted for.

> **Example 4:** Assume that all of the devices that τ_3 uses (i.e., d_2, d_3, and d_4) are single-request devices and that the I/O service for these devices is synchronous. Also assume that the client process becomes idle for the duration of the I/O service phase (i.e., interrupts are used to signal I/O completion).
>
> The components of execution for the input, processing, and output stages are:
>
> $$(C_{3,s1} + C_{3,d2}+2C_s + C_{3,d3}+2C_s) + C_{3,p} + (C_{3,s2} + C_{3,d4}+2C_s) + 2C_s$$
>
> There are three devices which cause process idle time and potentially two context switches for each. Thus, the execution time component of the inequality is $(C_3^{'}+6C_s)$. The inequality for this process is:
>
> $$\frac{C_1^{'}}{T_1} + \frac{C_2^{'}}{T_2} + \frac{(C_3^{'} + 6C_s)}{T_3} + \frac{X_3}{T_3} \leq 3(2^{1/3}-1)$$
>
> Note that X_k includes components from the I/O service phase of each device that process τ_3 uses. In the previous example, this component of time was included in the execution time term. (See page 9 for difference between the components of execution time for I/O operations with and without idle time.) Basically, the net effect of idle time on the schedulability of process τ_3 is additional context switching overhead.

Idle time has an effect on the blocking time components of higher priority processes.

If lower priority processes become idle and use interrupts as means of signalling I/O completion, then the period of non-preemptibility that starts with the interrupt is treated as blocking time for higher priority processes.

Idle time may also affect the blocking properties of the process that experiences inactivity. The priority ceiling protocol's "blocked at most once" property is preserved if a process is idle while a resource is locked. When the resource is locked, another process must have a priority that is strictly greater than the priority ceiling of all other locked resources in order to lock any resource. Consequently, lower priority processes will not be allowed to lock other resources while the client is idle.[12]

Idle time also affects the properties of PCP emulation (discussed in Section 3.1). Since clients use one resource at a time and then release it, there is no *hold and wait* condition and consequently deadlock is not a problem [7]. However, inactivity can allow queues to form. If queues are FIFO rather than prioritized, blocking time for higher priority processes will be increased.

> **Example 5:** This example is the same as the previous one except that in addition to devices d_2, d_3, and d_4, the I/O service to device d_5 also involves idle time. Once again we are faced with two problems in calculating the blocking term for τ_3: finding the various sources of blocking and determining the right function for combining the various forms of blocking.

[12] If, however, the client has not locked a resource when it becomes idle, the client is no longer protected by the priority ceiling protocol and a lower priority process may lock a resource that the inactive client will eventually need. In this case, the client may be blocked once for each time it idles, in addition to being blocked once before it idles. This situation may arise when the process is using a dedicated single-request device. Since the device is dedicated, mutual exclusion is not needed and thus the PCP does not come into play. One might entertain protecting the resource with a semaphore or monitor so that the PCP could be used to avoid multiple blocking.

One source of blocking to τ_3 is due to resource sharing:

$$max(\ C_{4,d4},\ C_{5,s1},\ C_{5,s2}\ )$$

Another source of blocking is due to the interrupt associated with device d_5:

$$Cpt(C_{5,d5})$$

Recall that $Cpt(C_{5,d5})$ is the non-preemptible duration of the I/O-completion phase of process τ_5's I/O operation using device d_5. In order for this interrupt to cause blocking, d_5 must be locked by τ_5 when τ_3 preempts. Since the use of the synchronous paradigm implies that only one device is locked by any given client at one time, we know that s_1 and s_2 are not locked when d_5 is locked. For this reason, simply adding together the above two blocking terms is overly pessimistic. For example, if we set the blocking term to be:

$$B_3 = max(\ C_{5,s1},\ C_{5,s2},\ C_{4,d4}) + Cpt(C_{5,d5})$$

and

$$max(\ C_{4,d4},\ C_{5,s1},\ C_{5,s2}\ ) = C_{5,s1}$$

then

$$B_3 = C_{5,s1} + Cpt(C_{5,d5})$$

However, since s_1 and d_5 cannot be simultaneously locked, the blocking contributions are not additive.

On the other hand, consider the following case: τ_5 locks d_5; τ_5 is then preempted by τ_4, which locks d_4; d_4 is in turn preempted by τ_3. At this point, d_5 completes, resulting in an interrupt and consequently blocking time for τ_3. When τ_3 resumes it attempts to lock d_4 and is blocked again. The blocking term for this scenario is:

$$Cpt(C_{5,d5}) + C_{4,d4}$$

Therefore, the blocking term for τ_3 is:

$$B_3 = max\Big(max(C_{4,d4},\ C_{5,s1},\ C_{5,s2}),\ (Cpt(C_{5,d5}) + C_{4,d4})\Big)$$

which can be reduced to:

$$B_3 = max\Big(C_{5,s1},\ C_{5,s2},\ (Cpt(C_{5,d5}) + C_{4,d4})\Big)$$

The point of the exercise is to explicate the factors that contribute to blocking and to show how to reason about combining the various factors.

Idle time can also affect lower priority processes. The idle time of a higher priority process offers a lower priority process an opportunity to execute. However, additional context-switching overhead due to this inactivity is one cost that weighs against the benefit of less preemption time. A more subtle cost is the cost due to deferred execution. The deferred execution effect due to the inactivity of the higher priority process must be accounted for in the schedulability inequality of lower priority processes.

Example 6: In this example let device d_1 be the only resource that involves idle time. The purpose of this example is to analyze the tradeoffs in determining the schedulable utilization of process τ_3 when a higher priority process becomes inactive.

Process τ_2 is the only process that uses device d_1. The inequality for process τ_2 has to account for idle time and hence X_2 will include a term $Srv(C_{2,d1})$. The inequality for process τ_3 does not have to include $Srv(C_{2,d1})$ as preemption time, but additional context switching must be accounted for, and the effects of deferred execution must be included. The inequality that models this situation is:

$$\frac{C_1'}{T_1} + \frac{(C_2' + 2C_s)}{T_2} + \frac{C_3'}{T_3} + \frac{B_3 + D}{T_3} \leq 3(2^{1/3} - 1)$$

D is the additional term that is needed to model the deferred execution effect due to process τ_2. B_3 is blocking due to lower priority processes. Recall from Section 3.1 that the deferred execution effect can be modeled by adding a term to lower priority processes to account for the effect. The term is the minimum of the amount of execution time that is deferred and the duration of the period of inactivity. Referring to Figure 4-3, it can be seen that the term in this case is:

$$D = min\left(Srv(C_{2,d1}),\ (C_2' + 2C_s) - (St(C_{2,d1}) + 2C_s)\right)$$

which reduces to

$$D = min\left(Srv(C_{2,d1}),\ C_2' - St(C_{2,d1})\right)$$

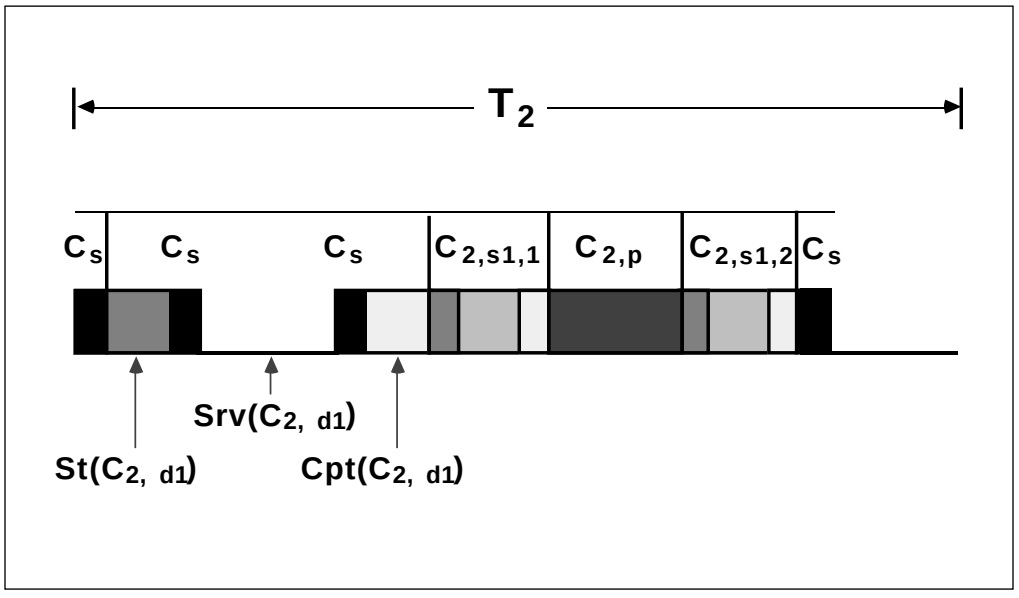


Figure 4-3: Deferred Execution

Now we will assess the schedulability benefits of process τ_2's idle time for process τ_3.

First assume that the idle time is relatively short compared to the deferred execution time:

$$min\left(Srv(C_{2,d1}),\ C_2^{'} - St(C_{2,d1})\right) = Srv(C_{2,d1})$$

Without idle time (i.e., if polling is used), the inequality for process τ_3 is:

$$\frac{C_1^{'}}{T_1} + \frac{(C_2^{'} + Srv(C_{2,d1})}{T_2} + \frac{C_3^{'}}{T_3} + \frac{B_3}{T_3} \leq 3(2^{1/3} - 1)$$

With idle time, the inequality for process τ_3 is:

$$\frac{C_1^{'}}{T_1} + \frac{(C_2^{'} + 2C_s)}{T_2} + \frac{C_3^{'}}{T_3} + \frac{B_3 + Srv(C_{2,d1})}{T_3} \leq 3(2^{1/3} - 1)$$

From the above inequalities we can see that if the following inequality is satisfied, then the schedulable utilization of process τ_3 has improved due to idle time of process τ_2.

$$\frac{Srv(C_{2,d1}) - 2C_s}{T_2} > \frac{Srv(C_{2,d1})}{T_3}$$

Basically the inequality tells us that if context switching is small relative to idle time, then idle time is beneficial to the lower priority process.

Now assume that the idle time is large relative to the execution time that is deferred:

$$min\left(Srv(C_{2,d1}),\ C_2^{'} - St(C_{2,d1})\right) = C_2^{'} - St(C_{2,d1})$$

The following inequality governs the tradeoff in this case:

$$\frac{Srv(C_{2,d1}) - 2C_s}{T_2} > \frac{C_2^{'} - St(C_{2,d1})}{T_3}$$

The inequality tells us that if idle time is significantly greater than the deferred execution time (i.e., idle time minus context switching overhead is greater than deferred execution time), then idling is beneficial to the lower priority process. In both cases, analysis confirms intuition.

4.2. Asynchronous I/O

The previous sections analyze the effects of non-preemptibility and idle time. In particular, the circumstances under which lower priority processes could increase their schedulable utilization by taking advantage of idle time in higher priority processes are examined. The essence of this section is to explore how a process can take advantage of its own idle time to increase its schedulable utilization. We first investigate asynchronous I/O in the context of devices that can only handle one I/O request at a time, the so-called single-request devices.

4.2.1. Single-Request Devices

Total process idle time can be reduced by allowing the process to perform other work while the I/O service is in progress, thus effectively increasing its own schedulable utilization. Consider Figures 4-4 and 4-5 for an illustration of the differences between synchronous and asynchronous idle time. First notice that in the synchronous case, all idle times contribute in an additive manner to execution time. The idle time component in the process τ_k's inequality is:

$$Idle\ Time = \sum_{r \in Dev(\tau_k)} Srv(C_{k,r})$$

Consider the asynchronous paradigm (Figure 4-5). Idle time for the input stage cannot be any longer than the maximum idle time for all of the input operations. The same is true for the output stage. Therefore, the worst-case idle time for the asynchronous paradigm is:

$$Worst\text{–}Case\ Idle\ Time = max(\ Srv(C_{k,r}\ ) \mid r \in\ InpDev(\tau_k)) + max(\ Srv(C_{k,r}\ ) \mid r \in\ OutDev(\tau_k))$$

Idle time can be further reduced by placing I/O requests involving CPU-dependent devices and/or shared data after I/O requests that involve idle time. The idea is to attain maximal CPU utilization during idle time.

Also notice that the asynchronous paradigm offers the opportunity to totally eliminate idle time from the output stage. Effectively, the I/O-completion phase of all output operations can be viewed as a check for successful I/O completion. This could easily be checked at the beginning of the following period as shown in Figure 4-6. An implementation paradigm for the client process performing synchronous I/O is shown in Figure 4-7 and for the two asynchronous alternatives in Figures 4-8 and 4-9.

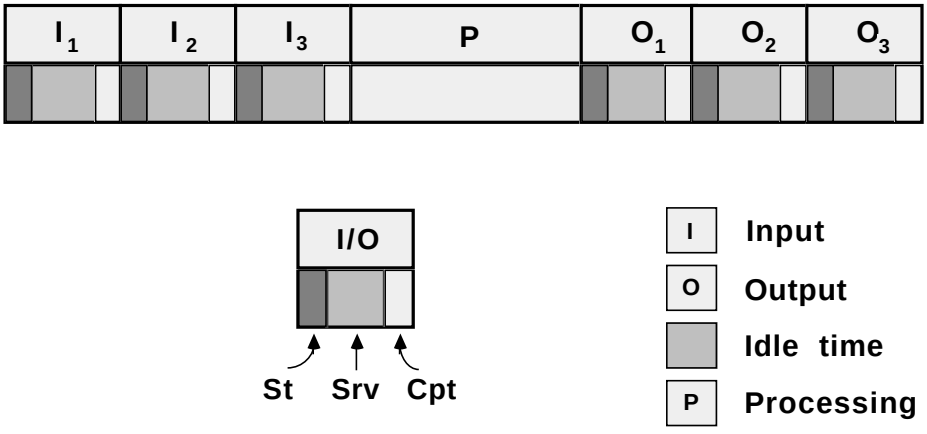


Figure 4-4: Synchronous Idle Time

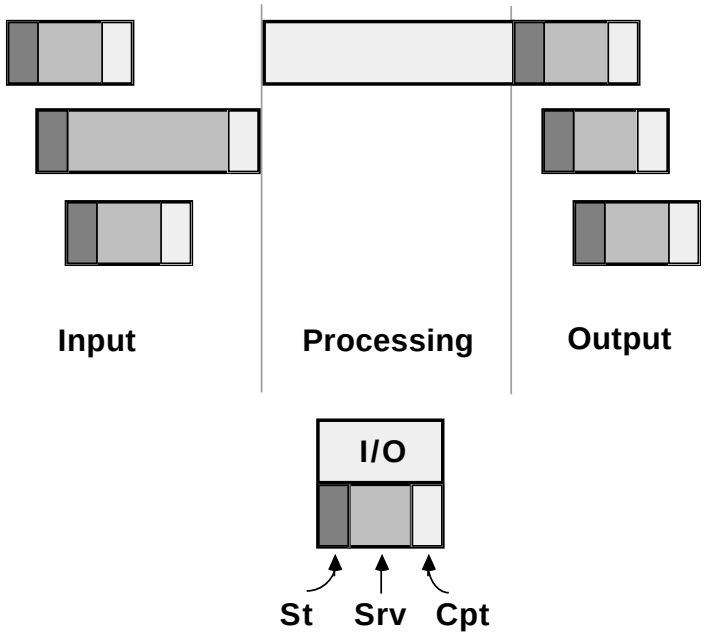


Figure 4-5: Asynchronous Idle Time

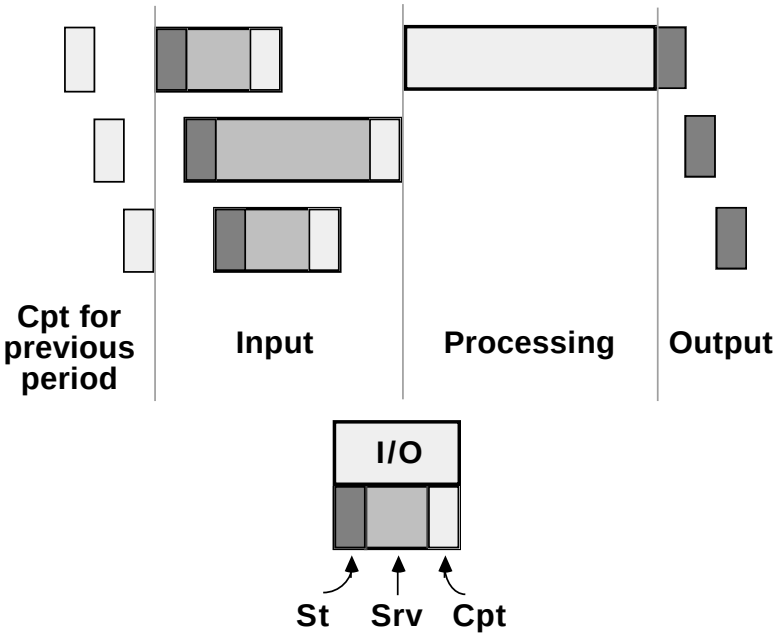


Figure 4-6: Optimized Asynchronous Idle Time

While asynchronous I/O paradigms allow total idle time to be reduced (when compared with synchronous paradigms), the blocking that the process causes to higher priority processes due to interrupts is worse for the asynchronous paradigms than for the synchronous paradigms. Consider the synchronous case for a moment. A lower priority process employing synchronous I/O will only have one outstanding I/O request at any given time. A higher priority process suffers blocking when it preempts the lower priority process while an I/O request is outstanding, since the device interrupts the CPU to signal the completion of the lower priority I/O operation while the higher priority process is still executing. Therefore, in the synchronous case the blocking contribution for higher priority processes due to interrupts related to process τ_k's I/O is:

$$max(Cpt(C_{k,d}) \mid d \in Dev(\tau_k))$$

In the asynchronous case there can be multiple outstanding I/O requests when a higher priority process preempts. The worst case occurs when the lower priority process is preempted after it has issued all of its requests for input or all of its requests for output. The equivalent blocking contribution for higher priority processes in this case is:

$$max\left(\sum_{r \in InpDev(\tau_k)} \sum_{l} Cpt(C_{k,r,l}) \, , \sum_{r \in OutDev(\tau_k)} \sum_{l} Cpt(C_{k,r,l}) \right)$$

```
task body Client is
begin
   IO_Services_Device_a.Read;    -- Input Stage
   IO_Services_Device_b.Read;
   IO_Services_Device_c.Read;

   Processing_Stage;             -- Processing Stage

   IO_Services_Device_1.Write;   -- Output Stage
   IO_Services_Device_2.Write;
   IO_Services_Device_3.Write;
end Client;
```

Figure 4-7: Synchronous I/O: Client

Implementation paradigms for asynchronous I/O require careful consideration to ensure that the benefits of the priority ceiling protocol are preserved.[13] Given the implementation paradigms for client processes for the synchronous and asynchronous cases as shown Figures 4-7 and 4-8 respectively, we now turn to the associated implementation paradigms for the monitor processes.

[13]In general, the implementation paradigms are illustrated using an Ada-like syntax but are not meant to be Ada-specific. However, in this section we couch our discussion specifically in terms of Ada, since the application of the priority ceiling protocol to Ada has already been defined in [2].

```
task body Client is
begin
   IO_Services_Device_a.Asyn_Read;   -- Input Stage
   IO_Services_Device_b.Asyn_Read;
   IO_Services_Device_c.Asyn_Read;

   IO_Services_Device_c.Wait_Read( <Buffer> );
   IO_Services_Device_b.Wait_Read( <Buffer> );
   IO_Services_Device_a.Wait_Read( <Buffer> );

   Processing_Stage;                 -- Processing Stage

                                     -- Output Stage
   IO_Services_Device_1.Asyn_Write( <Buffer> );
   IO_Services_Device_2.Asyn_Write( <Buffer> );
   IO_Services_Device_3.Asyn_Write( <Buffer> );

   IO_Services_Device_3.Wait_Write;
   IO_Services_Device_2.Wait_Write;
   IO_Services_Device_1.Wait_Write;

end Client;
```

Figure 4-8: Asynchronous I/O: Client

```
task body Client is
begin
   IO_Services_Device_3.Wait_Write;  -- Finish Output Stage
   IO_Services_Device_2.Wait_Write;  --  from previous period
   IO_Services_Device_1.Wait_Write;

   IO_Services_Device_a.Asyn_Read;   -- Input Stage
   IO_Services_Device_b.Asyn_Read;
   IO_Services_Device_c.Asyn_Read;

   IO_Services_Device_c.Wait_Read( <Buffer> );
   IO_Services_Device_b.Wait_Read( <Buffer> );
   IO_Services_Device_a.Wait_Read( <Buffer> );

   Processing_Stage;                 -- Processing Stage

                                     -- Output Stage
   IO_Services_Device_1.Asyn_Write( <Buffer> );
   IO_Services_Device_2.Asyn_Write( <Buffer> );
   IO_Services_Device_3.Asyn_Write( <Buffer> );
end Client;
```

Figure 4-9: Optimized Asynchronous I/O: Client

Recall that we are assuming the devices are single-request devices and thus can handle only one outstanding request. Hence, the devices require mutually exclusive access. The `IO_Monitors` in Figures 4-1 and 4-2 enforce mutually exclusive access in the synchronous case. Notice that the PCP rules, as applied to monitor processes [2], will ensure the "blocked at most once property" in this case.

```
package IO_Services_Device_n is
   procedure Asyn_Read;
   procedure Wait_Read( <Buffer> );

   procedure Asyn_Write( <Buffer> );
   procedure Wait_Write;
end IO_Services;



package body IO_Services_Device_n is
   procedure Asyn_Read is
   begin
      IO_Monitor.Asyn_Read;
   end Asyn_Read;

   procedure Wait_Read( <Buffer> ) is
   begin
      IO_Monitor.Wait_Read( <Buffer> );
   end Wait_Read;

   procedure Asyn_Write( <Buffer> ) is
   begin
      IO_Monitor.Write( <Buffer> );
   end Asyn_Write;

   procedure Wait_Write is
   begin
      IO_Monitor.Wait_Write;
   end Wait_Write;

end IO_Services;
```

Figure 4-10: Asynchronous I/O: Interface

Asynchronous I/O requires an implementation paradigm that facilitates mutual exclusion in a manner similar to that shown via the synchronous monitor (Figure 4-2), but must allow the client process to start the I/O operation and then have control returned to perform other work. One option for an implementation paradigm is shown in Figures 4-10 and 4-11. However, this structure violates Ada coding restrictions for server tasks outlined in [2]. The coding restrictions developed in [2] were motivated by the need to preserve the desirable properties of the priority ceiling protocol, which was originally defined in terms of rules for locking binary semaphores [10]. In order to use asynchronous I/O and continue to benefit from the desirable properties of the PCP, the asynchronous I/O services must be implemented in a manner that is consistent with the PCP. One approach is to incorporate a semaphore into the asynchronous I/O services. Specifically, implement the `Asyn_Read (Asyn_Write)` so that P operation is performed in addition to the Start I/O request and implement `Wait_Read (Wait_Write)` so that a V operation is performed during I/O Completion. The semaphore operations that are embedded in the I/O services must conform to the semaphore locking rules of the PCP.

```
task body IO_Monitor is
begin
  loop
      select
        accept Read do
            Start IO;
        end Read;

        Wait for I/O Interrupt;

        accept Wait_Read( <Buffer> )  do
            Data movement or pointer manipulation;
        end Wait_Read;

      or
        accept Write( <Buffer> )  do
            Data movement or pointer manipulation;
            Start IO;
        end Write;

        Wait for I/O Interrupt;

        accept Wait_Write;

      end select;
  end loop;
end IO_Monitor;
```

Figure 4-11: Asynchronous I/O: Monitor

4.2.2. Considerations for Multi-Request Devices

There are several noteworthy considerations for devices that support multiple outstanding requests. One consideration is that the implementation paradigm for supporting this type of device is slightly more complicated. This is discussed Appendix A.

It is also important to know the mechanism the device uses to manage multiple requests. A simple model of this type of device involves a simple processor and a queue manager. The device queues requests from the CPU and works to empty the queue. The issue of concern is the queuing discipline. If the queue is a FIFO queue, low priority requests may be serviced before higher priority requests and consequently the device has introduced another source of blocking. FIFO queues in devices can be a serious bottleneck for high priority tasks.

4.2.3. Considerations for Emulating Multi-Request Devices

Blocking time associated with accessing a single-request device can be reduced by emulating a multi-request device. Consider the client process illustrated in Figure 4-8 and the associated monitor process in Figure 4-11. Once this client process issues its request to start the first read operation using device **a**, the device is locked until both the data has been moved and the client process issues a call to `Wait_Read` for device **a**. The worst-case

blocking time for a higher priority client process that shares the device is the duration of time from the `Asyn_Read` to the `Wait_Read`. However, the device may have completed data movement before the lower priority client gets to the point in its processing where it can execute the call to `Wait_Read`. If this is the case, the higher priority client is blocked longer than necessary. This points out a fundamental difference between using devices and memory-resident shared resources. When using devices that operate physically concurrent with the CPU, the resource may be ready for the next client before the current client is ready for the results of the I/O operation. Emulating a multi-request device by creating a queue of I/O requests allows the high priority client to use the single-request device as soon as data movement is completed.

In this case, an application can submit multiple asynchronous requests for I/O without having to lock the device (i.e., only having to the lock the device for the duration of the request, as opposed to the duration of the I/O service). This abstraction also requires a queue of outstanding I/O requests. However, in this case the queue is managed by the executive. This raises two important concerns:

- Once again, a FIFO queuing discipline will result in blocking.
- Even if a priority queue is used, queue management may result in blocking if it is performed within the executive at effectively a higher priority.

4.2.4. Pipelining of I/O Requests

Pipelining is used to take maximal advantage of idle time at the cost of introducing latency in the results. The sequential paradigms require that the input stage complete before the processing stage commences and that the processing stage complete before the output stage commences. Pipelining allows these stages to overlap. For example, the processing stage can take advantage of idle time in the input stage.

In order to allow the processing stage to capitalize on the idle time in the input stage, processing must commence before input is completed. This means that the processing performed during a given period must use input collected during the previous period, as shown in Figure 4-12. This is known as *double-buffered input*.

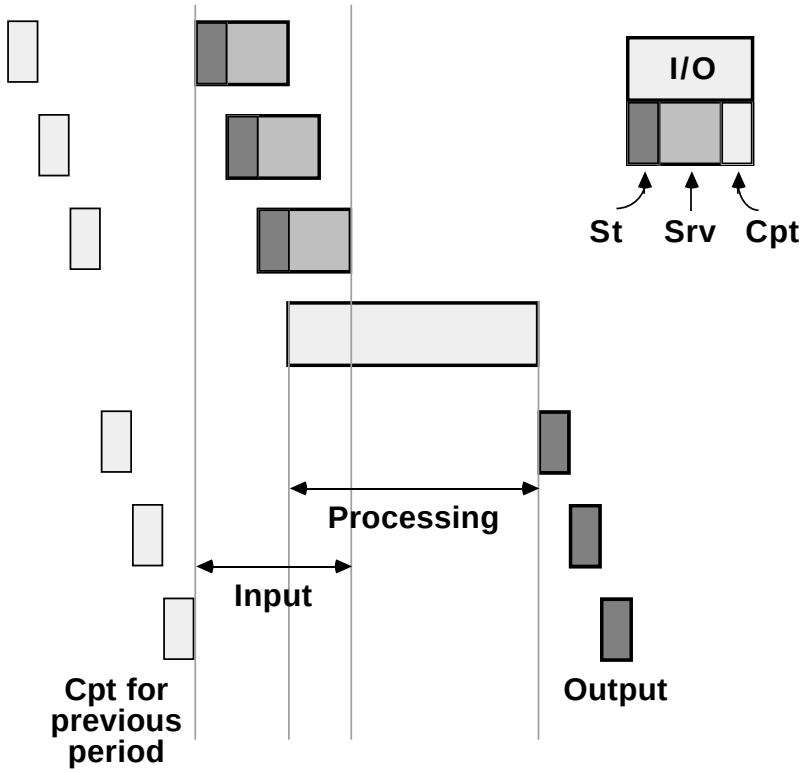


Figure 4-12: Pipelining

A consequence of this paradigm is latency. The output generated during any given period reflects the input from one period before it. This is illustrated in Figure 4-13. The input from period i-1, denoted as (Inp i-1), is processed in period i, denoted as (Proc i-1), and is output during period i, denoted as (Out i-1). Notice that even though the data that is output is essentially one period old, new output is generated every period. Therefore (if the latency can be tolerated), this paradigm is suitable for generating periodic output.

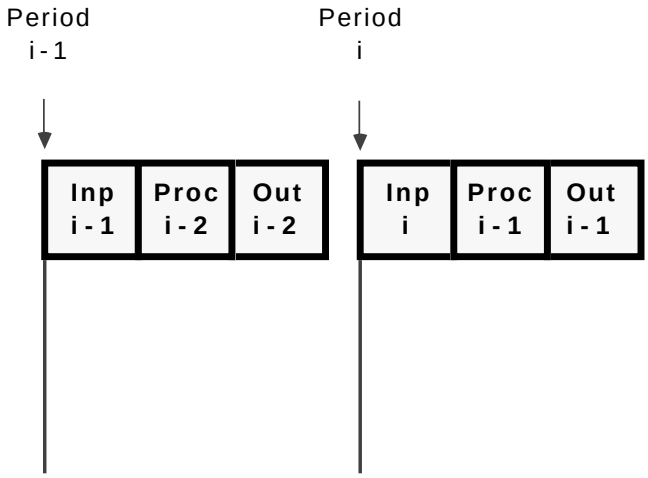


Figure 4-13: Latency Due to Pipelining

This paradigm effectively reduces the deferred execution effects of service time to zero; all of the idle time occurs after all of the non-idle time. If one avoids waiting for I/O completions at the end of the period and instead checks for completion at the beginning of the following period, this paradigm avoids the context switching penalty that other paradigms pay for idle time. This is illustrated in the sample client in Figure 4-14. Since there is no idle time, there also is no deferred execution penalty for lower priority tasks.

```
task body Client is
begin
-- Assuming this is period i:

-- Gather data from Asyn_Read initiated in period i-1
     IO_Services_Device_a.Wait_Read(<Inp i-1>);
     IO_Services_Device_b.Wait_Read(<Inp i-1>);
     IO_Services_Device_c.Wait_Read(<Inp i-1>);

-- Confirm completion of  Asyn_Write initiated in period i-1
     IO_Services_Device_3.Wait_Write;
     IO_Services_Device_2.Wait_Write;
     IO_Services_Device_1.Wait_Write;

-- Initiate Asyn_Read for period i
     IO_Services_Device_a.Asyn_Read;
     IO_Services_Device_b.Asyn_Read;
     IO_Services_Device_c.Asyn_Read;

-- Initiate processing using data gather above
     Processing_Stage( <Inp I-1>, <Out I-1>);

-- Initiate Asyn_Write using data  from above processing
     IO_Services_Device_1.Asyn_Write(<Out I-1>);
     IO_Services_Device_2.Asyn_Write(<Out I-1>);
     IO_Services_Device_3.Asyn_Write(<Out I-1>);

end Client;
```

Figure 4-14: Asynchronous I/O with Pipelining: Client

This paradigm also results in a blocking penalty that is due to interrupts. Recall that the blocking time for the asynchronous-sequential paradigm was:

$$max\left(\sum_{r \in InpDev(\tau_k)} \sum_{l} Cpt(C_{k,r,l}) \, , \sum_{r \in OutDev(\tau_k)} \sum_{l} Cpt(C_{k,r,l}) \right)$$

The blocking penalty for higher priority tasks in this case is:

$$\sum_{r \in Device(\tau_k)} \sum_{l} Cpt(C_{k,r,l})$$

5. Summary and Conclusion

This report illustrates how the principles of rate monotonic scheduling theory can be methodically applied to variations of synchronous and asynchronous I/O paradigms. We have varied the characteristics of **synchronous I/O** operations to explore:

1. **Effects of non-preemptibility.** Non-preemptibility is a source of blocking. When calculating worst-case blocking effects due to non-preemptibility, one can use a "blocked at most once" rule like that used for the priority ceiling protocol.

2. **Effects of idle time.** Idle time potentially affects the schedulability of the idling process as well as higher and lower priority processes. The scheduling inequality for the process itself must include a term to account for this gap in execution and additional context switching. Higher priority processes will be affected by interrupts that signal I/O completion. Interrupts on behalf of an idling process represent blocking time to higher priority processes. Lower priority processes must account for the deferred execution effect. A lower priority process benefits from a higher priority process's idle time if one of the following conditions is true:

    - Idle time is small relative to the execution time that is deferred and context switching time is small relative to the idle time.

    - Idle time is significantly larger than the execution time that is deferred.

Asynchronous I/O was then introduced as a means of reducing a process's idle time. We explored asynchronous I/O in the context of:

1. **Single-request devices.** We explored two paradigms for implementing mutual exclusion for this type of device. The first mechanism was very similar to a semaphore and required locking rules that adhered to the priority ceiling protocol. This paradigm requires that the client process retain the lock for the duration of the I/O operation. However, it is possible for the I/O operation to complete before the client process can reach the point in its execution where it can release the lock, thus locking out other potential clients longer than is necessary. Emulating multi-request devices represents a paradigm that avoids this problem.

2. **Multi-request devices.** One must be aware of the discipline used to queue multiple requests. FIFO queues in software and in devices can be a serious bottleneck.

3. **Pipelining.** This technique further reduces idle time at the cost of introducing latency into the results. Also, a price paid for reducing idle time using asynchronous paradigms is increased blocking to higher priority process due to interrupts.

We have also explored the notion of incrementally constructing a schedulability model of a

real-time system, where the schedulability model is a mathematical model of the timing and concurrency structure of the system. A schedulability model can be built incrementally by considering each process and determining all of the factors that influence its schedulability and how it influences the schedulability of other processes.

There are several areas that were not discussed. We assumed all processes were periodic. Extending the general model of I/O-related processing to incorporate aperiodic events is natural. We also feel that the techniques presented in this report are naturally extensible to the situation where the processing stage is dispersed throughout a process's execution.

We encourage the reader to apply the presented analysis techniques to problems not explicitly addressed in this report.

Acknowledgements

The authors would like to express appreciation to the following individuals for their insightful comments: Mark Borger, Mike Gagliardi, John Goodenough, Keith Kohout, B. Craig Meyers, Lui Sha and to Lisa Jolly for her assistance in creating figures for the report.

Appendix A: Implementation Paradigm for Multi-Request Devices

Multi-request devices, by definition, support multiple outstanding I/O requests. Since there can be multiple outstanding I/O requests, a mechanism is needed for associating an I/O completion with the corresponding client process that started the I/O operation. The mechanism used is similar to the mechanism used by a pizza shop. The customer (analogous to the client process) places his or her order (analogous to making an I/O request) and receives a ticket with a number (analogous to the I/O identification number returned to the client process), which is called when the pizza is ready. The customer either waits for his or her number to be called or leaves the pizza shop for a short period of time and then returns to present his or her number and ask if the pizza is ready.

A procedural interface for this type of I/O paradigm is shown in Figure A-1. When `Asyn_Read` and `Asyn_Write` are called to request an I/O operation, they return to the client an I/O identification number (ID). When `Wait_Read` and `Wait_Write` are called to wait for completion of an I/O operation, they require use of the I/O ID.

```
package IO_Services is
  subtype ID_type is range 1..Max_IDs;
  type Buffer_Type is ...

  procedure Asyn_Read(ID: out ID_type );
  procedure Asyn_Write(ID: out ID_type; Buffer: out Buffer_Type );

  procedure Wait_Read(ID: in ID_type; Buffer: in Buffer_Type );
  procedure Wait_Write(ID: in ID_type);
end IO_Services;
```

Figure A-1: Multi-Request Interface

Figure A-2 illustrates that procedures `Asyn_Read` and `Asyn_Write` are simply procedural interfaces to the associated entries of the monitor process shown in Figure A-3.

The monitor reserves the I/O ID through a call to `Reserve_Completion_ID` and starts the I/O operation in a critical section. Since multi-request devices do not require mutually exclusive access for the entire duration of an I/O operation, mutual exclusion is provided for only the start I/O phase. This is illustrated in Figure A-3.

`Reserve_Completion_ID` searches the `IO_Waiter` array shown in Figure A-4 for an element that satisfies `IO_Waiter(ID).Reserved = FALSE`. The I/O ID is simply an index into an array of records. There is a one-to-one relationship between tickets in the above analogy, I/O ID's, and elements in the array of records.

```
procedure Asyn_Read(ID: out ID_type ) is
begin
   IO_Monitor.Read( ID );
end Asyn_Read;


procedure Asyn_Write(ID: out ID_type ) is
begin
   IO_Monitor.Write( ID );
end Asyn_Write;


procedure Wait_Read(ID: in ID_type; Buffer: out Buffer_Type ) is
begin
  IO_Waiter(ID).Wait_For_IO_Completion( Buffer_Pointer );
  Release_Completion_ID( ID );
end Wait_Read;


procedure Wait_Write(ID: in ID_type ) is
begin
  IO_Waiter(ID).Wait_For_IO_Completion( Buffer_Pointer );
  Release_Completion_ID( ID );
end Wait_Write;
```

Figure A-2: Multi-Request Procedural Interface

```
task IO_Monitor is
begin
  loop
      select
        accept Read( ID: out ID_type ) do
            Reserve_Completion_ID( ID );
            Start_IO_for_Read;
        end Read;
      or
        accept Write( ID: out ID_type ) do
            Reserve_Completion_ID( ID );
            Start_IO_For_Write;
        end Write;
      end select;
  end loop;
end IO_Monitor;
```

Figure A-3: Multi-Request Monitor for Requesting I/O

The I/O ID is then used by `Wait_Read` and `Wait_Write`, as shown in Figure A-2, as a means to indicate the I/O operation for which it is waiting. An interrupt service routine shown in Figure A-5 also uses the I/O ID to notify the right client of I/O completion, as shown in Figure A-4.

Before returning to the client process, `Wait_Read` and `Wait_Write` call `Release_Completion_ID` to release the identifier for subsequent use.[14]

[14]Note that if this paradigm were implemented as part of the executive or runtime system, information such as process-ID would be readily available, obviating the need to explicitly pass an ID back to the client process.

```
package body IO_Services is

  type Buffer_Pointer_Type is ...

  task type IO_Wait_Task_Type is
    entry IO_Complete
          ( Buffer_Pointer: in Buffer_Pointer_Type);
    entry Wait_for_IO_Completion
          ( Buffer_Pointer: out Buffer_Pointer_Type);
  end IO_Wait_Task_Type

  task body IO_Wait_Task_Type is
  begin
    loop
        accept IO_Complete
               ( Buffer_Pointer: in Buffer_Pointer_Type);

        accept Wait_For_IO_Completion
               ( Buffer_Pointer: out Buffer_Pointer_Type ) do
    end loop;
  end IO_Wait_Type;


  type IO_Wait_Type is
    record
        Reserved : BOOLEAN;
        IO_Wait_Task : IO_Wait_Task_Type;
    end record;


  type IO_Wait_Array_Type is array(ID_type) of IO_Wait_Type;
  IO_Waiter : IO_Wait_Array_Type;


  procedure Reserve_Completion_ID(ID: out ID_type) is
  begin
    Find a Completion_ID;
    IO_Waiter(ID).Reserved := TRUE;
  end Reserve_Completion_ID;


  procedure Release_Completion_ID(ID: in ID_type) is
  begin
    IO_Waiter(ID).Reserved := FALSE;
  end Release_Completion_ID;


  -- See Figure A-2 for other procedures.
  -- See Figure A-3 for the monitor task.
  -- See Figure A-5 for the interrupt service routine.

end IO_Services;
```

Figure A-4: Multi-Request Completion_ID Management

```
task body Interrupt_Service_Routine is
begin
  loop
      accept Interrupt do
        Determine ID and Buffer_Pointer of completed I/O.
      end Interrupt;

      IO_Waiter(ID).IO_Complete( Buffer_Pointer );
  end loop;
end Interrupt_Service_Routine;
```

Figure A-5: Interrupt Service Routine

Appendix B: Figures for Example Problem

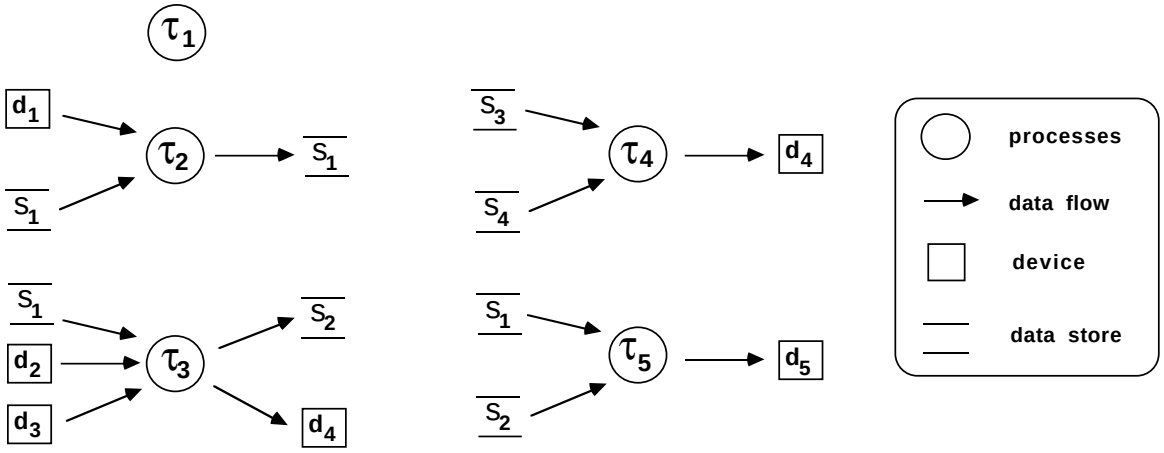


Figure B-1: Process/Resource Relationships in the Example Problem

	d_1	d_2	d_3	d_4	d_5	s_1	s_2	s_3	s_4
τ_1									
τ_2	x					x			
τ_3		x	x	x		x	x		
τ_4				x				x	x
τ_5					x	x	x		

Table B-1: Process/Resource Relationships in the Example Problem

References

1. Borger, M. W., Klein, M. H., Veltre, R. A. "Real-Time Software Engineering in Ada: Observations and Guidelines". *Software Engineering Institute Technical Review* (1988).

2. Goodenough, J. B., Sha, L. The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks. Proceedings of the International Workshop of Real-Time Ada Issues, June, 1988.

3. Lehoczky, J. P., Sha, L. "Performance of Real-Time Bus Scheduling Algorithms". *ACM Performance Evaluation Review, Special Issue 14*, 1 (May, 1986).

4. Lehoczky, J. P., Sha, L., Strosnider, J. Enhancing Aperiodic Responsiveness in A Hard Real-Time Environment. IEEE Real-Time System Symposium, December, 1987.

5. Liu, C.L., Layland, J.W. "Scheduling Algorithms for Multi-Programming in a Hard-Real-Time". *Journal of the Association for Computing Machinery Vol. 20*, 1 (January 1973), pp. 46-61.

6. *VMEbus Products: Selector Guide.* Motorola, Inc., Tempe, Arizona, 1989.

7. Peterson, James L. and Silberschatz, Abraham. *Operating System Concepts.* Addison Wesley, 1986.

8. Rajkumar, R., Sha, L., Lehoczky, L. "On Countering The Effect of Cycle Stealing in A Hard Real-Time Environment". *IEEE Real-Time System Symposium* (December 1987).

9. Rajkumar, R., Sha, L., Lehoczky, J.P. Real-Time Synchronization Protocols for Multiprocessors. IEEE Real-Time Systems Symposium, December, 1988.

10. Sha, L., Rajkumar, R., Lehoczky, J. P. Priority Inheritance Protocols: An Approach to Real-time Synchronization. Tech. Rept. (CMU-CS-87-181), Department of Computer Science, Carnegie Mellon Uniersity, 1987.

11. Sha, L., Goodenough, J. B. Real-Time Scheduling Theory and Ada. Tech. Rept. CMU/SEI-89-TR-14, ADA211397, Software Engineering Institute, April 1989. Also in *Computer*, Vol. 23, No. 4, (April 1990), pp. 53-62, .

12. Sha, L., Rajkumar, R., Lehoczky, J.P., Ramamritham, K. Mode Changes in a Prioritized Preemptive Scheduling Environment. Tech. Rept. CMU/SEI-88-TR-34, ADA207544, Software Engineering Institute, November 1989.

13. Shaw, Mary. "Abstraction Techniques in Modern Programming Languages". *IEEE Software, Vol. 4*, (October 1984).

14. Sprunt, B., Sha, L., Lehoczky, J. P. "Aperiodic Task Scheduling for Hard Real-Time Systems". *The Journal of Real-Time Systems Vol. 1* (1989), pp. 27-60.

15. Stankovic, John A. "Misconceptions About Real-Time Computing". *Computer Vol. 21, No. 10* (October 1988), pp. 10-19.

Table of Contents

1. Introduction **1**
1.1. An Analytical Framework 2
1.2. Considerations for Input/Output 2

2. Processing Model **5**
2.1. Input/Output Paradigms 5
2.2. Models of Device Interactions 7
2.3. Notation and Terminology 8

3. Review of Rate Monotonic Theory **11**
3.1. Basic Results of Rate Monotonic Scheduling 11
3.2. Schedulability Models 16
3.3. Example Problem 18

4. Input/Output Paradigms **21**
4.1. Synchronous I/O 21
4.1.1. Preemptible Service 21
4.1.2. Considerations for Non-Preemptibility 24
4.1.3. Considerations for Idle Time 25
4.2. Asynchronous I/O 30
4.2.1. Single-Request Devices 31
4.2.2. Considerations for Multi-Request Devices 36
4.2.3. Considerations for Emulating Multi-Request Devices 36
4.2.4. Pipelining of I/O Requests 37

5. Summary and Conclusion **41**

Acknowledgements **43**

Appendix A. Implementation Paradigm for Multi-Request Devices **45**

Appendix B. Figures for Example Problem **51**

References **53**

List of Figures

Figure 2-1: General Model for a Process 5
Figure 2-2: Input Stage in Detail 6
Figure 3-1: Effects of Idle Time 16
Figure 3-2: Deferred Execution Effect 17
Figure 3-3: Process/Resource Relationships in the Example Problem 19
Figure 4-1: Synchronous Service with No Idle Time 22
Figure 4-2: Synchronous Service with Idle Time 26
Figure 4-3: Deferred Execution 29
Figure 4-4: Synchronous Idle Time 31
Figure 4-5: Asynchronous Idle Time 32
Figure 4-6: Optimized Asynchronous Idle Time 32
Figure 4-7: Synchronous I/O: Client 33
Figure 4-8: Asynchronous I/O: Client 34
Figure 4-9: Optimized Asynchronous I/O: Client 34
Figure 4-10: Asynchronous I/O: Interface 35
Figure 4-11: Asynchronous I/O: Monitor 36
Figure 4-12: Pipelining 38
Figure 4-13: Latency Due to Pipelining 39
Figure 4-14: Asynchronous I/O with Pipelining: Client 40
Figure A-1: Multi-Request Interface 45
Figure A-2: Multi-Request Procedural Interface 46
Figure A-3: Multi-Request Monitor for Requesting I/O 46
Figure A-4: Multi-Request Completion_ID Management 48
Figure A-5: Interrupt Service Routine 49
Figure B-1: Process/Resource Relationships in the Example Problem 51

List of Tables

Table 3-1: Process/Resource Relationships in the Example Problem 20
Table B-1: Process/Resource Relationships in the Example Problem 51