Executive Publisher *Don Fowley*
Associate Publisher *Dan Sayre*
Acquisitions Editor *Catherine Shultz*
Project Editor *Gladys Soto*
Editorial Assistant *Chelsee Pengal*
Marketing Manager *Chris Ruel*
Production Editor *Lea Radick*
Cover Designer *Michael St. Martine*
Cover Image *©Megumi Takamura/Dex Image/Getty Images*
Bicentennial Logo Design *Richard J. Pacifico*

This book was set in *Times Ten* by *Preparé* and printed and bound by *Hamilton Printing*. The cover was printed by *Phoenix Color*.

Copyright 2008 © John Wiley & Sons, Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the proper written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978)750-8400, fax (978)646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, (201)748-6011, fax (201)748-6008.

Library of Congress Cataloging-in-Publication Data

Peckol, James K.
Embedded system: a contemporary design tool / James K. Peckol.
p. cm.
ISBN 978-0-471-72180-2 (cloth)
1. Embedded computer system. 2. Object oriented methods (Computer science)
I. Title.
TK7895.E42P43 2008
004.16--dc22

2007017870

ISBN 978-0-471-72180-2

To order books or for customer service please call 1-800-CALL WILEY (225-5945).
Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

UNIVERSITY OF WASHINGTON LIBRARIES
NOV 0 8 2011.

Preface

INTRODUCING EMBEDDED SYSTEMS

Less than 150 years ago, shipping a new product, petroleum, down the Mississippi in barges was viewed with skepticism and fear of possible explosion. Fifty years later, electricity and electric lights were viewed as marvels of modern technology available only to a few. Another 50 years subsequent, someone suggested that the world would need at most three to four computers. Our views continue to change. Today we ship petroleum (still with concern) all over the world. Electricity has become so common that we are surprised if a switch is not available to turn on a light when we enter a room. The need for three to four computers has grown to hundreds of millions, perhaps billions, of installed computers worldwide.

This book presents a contemporary approach to the design and development of a kind of computer system that most of us will never see—those that we call embedded systems. The approach brings together a solid theoretical hardware and software foundation with real-world applications. Why do we need such a thing? A good question, let's take a look.

Today we interact with an embedded computer in virtually every aspect of our everyday life. From operating our car to riding an elevator to our office to doing our laundry or cooking our dinner, a computer is there, quietly, silently doing its job. We find the microprocessor—microcomputer—microcontroller—everywhere. Today these machines are ubiquitous. Like the electric light, without thought, we expect the antilock braking system in our car to work when we use it. We expect our mobile phone to operate like the stationary one in our home. We carry a computer in our pocket that is more powerful than the ones the original astronauts took into space.

Today we have the ability to put an increasingly larger number of hardware pieces into diminishingly smaller spaces. Software is no longer relegated to a giant machine in an air-conditioned room; our computer and its software go where we go. This ability gives engineers a new freedom to creatively put together substantially more complex systems with titillating functionality, systems that only science fiction writers thought of a few years ago. Such an ability also gives us the opportunity to solve bigger and more complex problems than we have ever imagined in the past—and to put those designs into smaller and smaller packages. These are definitely the most fun problems, the exciting kinds of things that we are challenged to work on. Okay, where do we begin?

The embedded field started almost by accident not too many years ago. In the early 70s Federico Faggin and many others at Intel and Motorola introduced the 4004, 8008, and 6800 microprocessors to the engineering world. Originally intended for use in calculators and in calculator-like applications, today, driven by evangelists like Faggin, the microprocessor has become a fundamental component of virtually everything we touch. With such widespread application, the ensured safety and reliability of such systems are absolutely essential.

The embedded systems field has grown virtually overnight from nonexistent several years ago to encompass almost every aspect of modern electrical engineering and computing

META-1026, p. 1
Meta v. Resonant
IPR2025-00823

11.7 Consider implementing an embedded system to control a traffic light as a foreground/background system. Each direction supports a left turn (right turn if traffic normally drives on the left hand side) and pedestrian-activated crosswalk control.

(a) Which tasks are foreground tasks?

(b) Which tasks are background tasks?

(c) Give a UML state diagram illustrating the behavior of the system during a change from north-south green to east-west green. Be certain to consider the operation with and without a left (right) turn and with and without a pedestrian.

(d) Give a UML sequence diagram for the events in part (c).

11.8 Repeat Problem 11.7 for a microwave cooker.

11.9 Repeat Problem 11.7 for a washing machine.

11.10 Repeat Problem 11.7 for a video-on-demand entertainment system for a large hotel.

11.11 Consider implementing an embedded system to control a traffic light as an RTOS-based system. Each direction supports a left turn (right turn if traffic normally drives on the left-hand side) and pedestrian-activated crosswalk control.

(a) Which tasks are the major tasks?

(b) Give a UML state diagram illustrating the behavior of the system during a change from north-south green to east-west green. Be certain to consider the operation with and without a left (right) turn and with and without a pedestrian.

(c) Give a UML sequence diagram for the events in part (c).

11.12 Repeat Problem 11.11 for a microwave cooker.

11.13 Repeat Problem 11.11 for a washing machine.

11.14 Repeat Problem 11.11 for a video-on-demand entertainment system for a large hotel.

11.15 Provide a UML class diagram for a task control block (TCB). Implement the design using a C struct data structure.

11.16 Design a method that would enable the dynamic allocation and deallocation of TCBs as tasks are created or terminated.

11.17 Modify the design in Example 11.1 to support a dynamic number of tasks in the task queue without using malloc and free (C) or new and delete (C++) while retaining the array as the queue container.

11.18 Provide a UML class diagram for a task queue that supports the dynamic insertion and deletion of tasks.

11.19 Implement the task queue specified in Example 11.1 to use a doubly linked list as the underlying data type for the queue container.

11.20 Combine the subsystems in Problem 11.16 and Problem 11.19.

11.21 Modify the design of the TCB in Problem 11.15 to support a task priority number in the range of {0–9}. Assume 0 is the highest and 9 the lowest priority.

Incorporate the modified TCB design into the task queue design in Problem 11.19. Modify the access method to always return the highest priority task.

11.22 Modify the design of the TCB in Problem 11.15 to support the inclusion of an estimate of execution time number in the range of {0–99}.

Incorporate the modified TCB design into the task queue design in Problem 11.19. Modify the access method to always return the shortest task.

11.23 Give a high-level description of how the system in Figure P11.22 works. You should not need more than 10 lines.

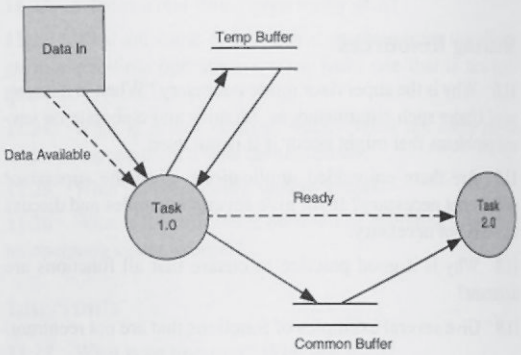


Figure P11.22

11.24 Write a C program to implement the design given in the data/control flow diagram in Problem 11.23.

Chapter 12

Tasks and Task Management

THINGS TO LOOK FOR . . .

- The role of time in embedded designs.
- The definitions of reactive and time-based systems.
- The differences between preemptive and nonpreemptive systems.
- The need for effectively scheduling the use of the system CPU(s).
- The criteria for making scheduling decisions.
- Common scheduling algorithms.
- Real-time scheduling considerations.
- How scheduling algorithms might be evaluated.
- Methods for intertask communication.
- The critical section problem and several solutions.
- Methods for task synchronization.

12.0 INTRODUCTION

In the previous chapter we introduced some of the basic concepts and methods involved in controlling multitasking systems. We learned that foreground / background systems can be effective under real-time constraints and that the basic responsibilities of the operating system comprise task scheduling, intertask communication, and task dispatch. In addition, we introduced some of the issues associated with the context switch in preemptable systems.

In this chapter, we will examine the scheduling problem and intertask communication in greater detail. The resource management aspects of task scheduling and dispatch will be covered in the following chapter. We will open by continuing the discussion of time and the critical role it plays in the design of embedded applications by introducing the concepts of *reactive* and *time-based systems*. We will present and discuss various metrics for specifying and assessing a task schedule. We will then investigate several different scheduling algorithms and analyze task synchronization and intertask communication in some detail. The focus will be primarily from the perspective of either a kernel-based or more complete operating system-based control strategy.

reactive, time-based systems

META-1026, p. 3
Meta v. Resonant
IPR2025-00823

.1 TIME, TIME-BASED SYSTEMS, AND REACTIVE SYSTEMS

.1.1 Time

We have already briefly encountered time and the important role it plays in the design and execution of embedded applications. We will now explore that role in greater detail.

absolute, relative

We define two different measures of time: *absolute* and *relative*, based on what the measurement is referenced to. Absolute time is based on real-world time; relative time is measured with respect to some reference. Time is further qualified as either an *interval* or a *duration*; these are distinct. An *interval* is marked by specific start and end times; a *duration* is a relative time measure. Equal intervals must have the same start times and the same stop times; nonequal intervals can have the same duration. This difference is captured in Figure 12.0.

interval
duration

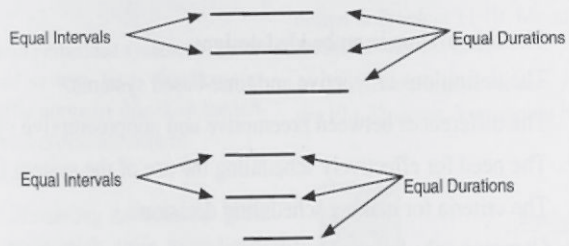


Figure 12.0 Equal Intervals and Equal Durations

.1.2 Reactive and Time-Based Systems

reactive, time based

Embedded systems are classified into two broad categories: *reactive* and *time based*. *Reactive* systems, as the name suggests, contain tasks that are initiated by some event that may be either internal or external to the system. An internal event may be an elapsed time or a temporal bound on data that has been exceeded. An external event is the recognition of a switch that has been activated or an external response to an internally generated command, for example. Typically, the initiating events are asynchronous to the normal activity of the system. Foreground/background systems are a good example of those classed as *reactive*.

time-based systems
absolute, relative
following an interval

Time-based systems are those systems whose behavior is controlled by time. Such a relationship can be *absolute*—an action must occur at a specific time; *relative*—an action must occur after or before some reference; or *following an interval*—an action must occur at a specified time with respect to some reference. The behavior in time-based systems is generally synchronous with a timing element of one form or another. Time-shared systems are a good example of those classed as *time based*.

periodic
aperiodic, periodic
execution times
jitter
delay

The relevance of time in embedded applications becomes clear when trying to schedule tasks and threads, that is, deciding when and how often each is executed. Tasks or threads that are initiated with repeating duration between invocations are called *periodic;* otherwise they are designated as *aperiodic*. A repeating duration is called the *period*. The time to complete a task is called the *execution time*.

In a periodic system, variation in the evoking event is called *jitter*. The time between the evoking event and the intended action is called the *delay*. When designing a system, each context in which it is anticipated that the system will be operating must be examined to determine the significance of jitter and delay with respect to specified time constraints.

hard, hard deadline

An action that must occur by a specified time is defined as *hard* or is said to have a *hard deadline*. A missed deadline in such cases is considered to be a partial or total system fail-

hard real-time

ure. A system is defined as *hard real-time* if it contains one or more tasks containing such constraints. Such systems may have other tasks that do not have temporal deadlines. The major focus, however, is on the hard deadlines.

soft real-time

Systems with relaxed time constraints are defined as *soft real-time*. Such systems may meet their deadlines on average. Soft real-time systems may be soft in several ways:

- Relaxation of the constraint that missing the deadline constitutes system failure. Such a system may tolerate missing the specific deadline provided some other deadline or timeliness constraint is met—the average throughput, for example.
- Evaluating the correctness of timeliness as a gradation of values rather than pass or fail.

firm real-time

Systems with tasks that have some relaxed constraints as well as hard deadlines are defined as *firm real-time*.

predictability

when, how

periodic

Real-time systems are those in which correctness demands timeliness. Most such systems carefully manage resources with respect to maintaining the *predictability* of timeliness constraints. Such predictability gives us a measure of accuracy with which one can state in advance *when* and *how* an action will occur. We elaborate by annotating the durations, events, jitter, and actions. Figure 12.1 illustrates a *periodic* system typical of a time-based design.

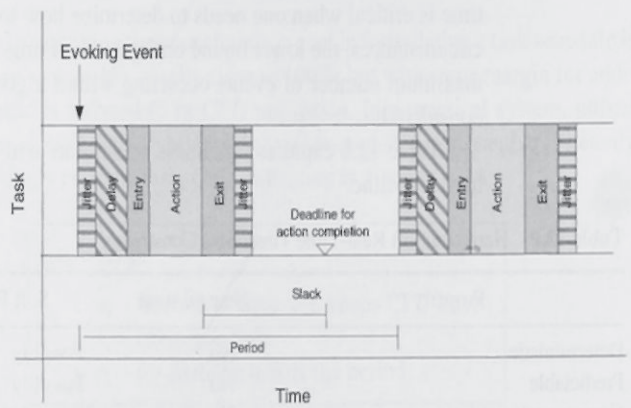


Figure 12.1 Task Activity in a Periodic Time-Based System

In the figure, the period of the recurrence of the tasks is defined. The evoking event occurs with respect to the start of the period. The first rectangle expresses the variation in the actual invocation with respect to the intended. Such jitter may arise from variations in the system's ability to respond to a timer expiring, for example. Once the event occurs, the second rectangle captures the delay in getting the task started. When the task begins to execute, the third rectangle accounts for any initialization or similar operations that must occur before the intended action takes place. The intended action occurs during the time indicated by the fourth rectangle. After the action completes, the fifth rectangle mirrors the entry actions with any necessary cleanup before the task completes. The sixth rectangle accounts for variation in exiting the task.

The diagram also marks the latest time at which the intended action could complete and still meet the time constraints on the period. The duration between the completion deadline and the start of the next cycle is equal to that between the end of the action and the end of the exit jitter.

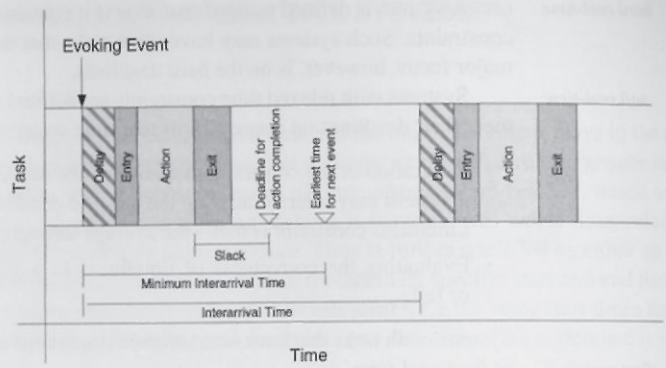


Figure 12.2 Task Activity in an Aperiodic Foreground/Background Design

aperiodic

Figure 12.2 illustrates an *aperiodic* system that is typical of a foreground/background design. Notice how the minimum and maximum times are specified.

The invocation of *aperiodic* tasks is not fixed in time—they are asynchronous to the operation of the core system. Thus, there can be no jitter because there is no expected time for the initiating event. The duration between such tasks is called *interarrival time*. Such a time is critical when one needs to determine how to schedule real-time tasks. Under such circumstances, the lower bound on interarrival time must be identified. Such things as the maximum number of events occurring within a given time interval may also need to be considered.

interarrival time

Table 12.0 captures timeliness constraints with respect to whether the task is soft or hard real-time.

Table 12.0 Hard and Soft Real-Time Timeliness Constraints

Property	Nonreal-time	Soft Real-time	Hard Real-time
Deterministic	No	Possibly	Yes
Predictable	No	Possibly	Yes
Consequences of late computation	No effect	Degraded performance	Failure
Critical reliability	No	Yes	Yes
Response dictated by external events	No	Yes	Yes
Timing analysis possible	No	Analytic (sometimes) stochastic simulation	Analytic, stochastic simulation

At this point, we should be sufficiently comfortable with some of the terminology that we can start to investigate the control of embedded systems in greater detail. We will begin with the problem of task scheduling.

.2 TASK SCHEDULING

How efficiently and effectively a task moves through the various queues along the control path following its arrival and how effectively and efficiently the CPU is utilized during such a movement establish the quality of the embedded design. An essential component of that control strategy is the algorithm used to schedule the allocation of the CPU.

In a multitasking system, the main objective is to have some process using the CPU at all times. Such a scheme maximizes the usage of that resource. Which task is running at any

specific time is based on a number of criteria. It is the scheduler's responsibility to ensure that the CPU is efficiently utilized and that the various jobs are executed in such an order as to meet any required constraints.

priority

When working with a scheduling algorithm, one must also consider the *priority* of the task. Priority is assigned by the designer and is based on a variety of different criteria. We will examine these shortly. Such criteria are used to resolve which task to execute when more than one is waiting and ready to execute. Tasks with higher priority execute preferentially over those with lower priority.

schedulable
deterministically schedulable

In a real-time context, a task that can be determined to always meet its timeliness constraints is said to be *schedulable*. A task that can be guaranteed to always meet all deadlines is said to be *deterministically schedulable*. Such a situation occurs when an event's worst case response time is less than or equal to the task's deadline. When all tasks can be scheduled, the overall system can be scheduled.

Scheduling decisions must be made during the design phase of the system development since such decisions involve trade-offs that affect and optimize the overall performance of the system. When the system specification stipulates hard deadlines, one must ensure that the implementing tasks and their associated actions can meet every deadline. Soft deadlines naturally give more flexibility.

12.2.1 CPU Utilization

CPU Utilization

In addition to satisfying time constraints, a goal in formulating a task schedule is to keep the CPU as busy as possible, ideally close to 100%, but with some margin for additional tasks. Such a metric is referred to as *CPU utilization*. In a practical system, utilization should range between 40% for a lightly loaded system and 90% for one that is heavily loaded.

For a single periodic task, CPU utilization is given as

$$u_i = e_i / p_i \tag{12.0}$$

u_I fraction of time task keeps CPU busy
e_i execution time
p_i for periodic task is the period

One can express a similar relationship for aperiodic tasks.

CPU utilization information can be used in conjunction with a sequence diagram to aid in assessing when each of the tasks can and needs to run.

12.2.2 Scheduling Decisions

Two key elements of real-time design, repeatability and predictability, are absolutely essential in the context of hard deadlines. To ensure predictability, one must completely understand and define the timing characteristics of each task and properly schedule those tasks using a predictable scheduling algorithm. The first step in developing a robust schedule is knowing when a scheduling decision must be made.

Scheduling decisions are made under the following four conditions:

running, waiting
running, ready
waiting, ready

1. A process switches from the *running* to the *waiting* state—initiated by an I/O request.
2. A process switches from the *running* to the *ready* state—when an interrupt occurs.
3. A process switches from the *waiting* to the *ready* state—the completion of I/O activity.
4. A process terminates.

META-1026, p. 7
Meta v. Resonant
IPR2025-00823

Asynchronous Interrupt Event Driven

One of the simplest scheduling schemes is asynchronous interrupt event driven. Certainly, the asynchronous nature of the scheme calls into question the use of the word "schedule." Under such an approach, the system is constrained to operate in a basic one-line infinite loop until an interrupting event occurs, as is illustrated in the code fragment shown in Figure 12.3. As such, the design is a special case of the foreground/background model. In this case, the design has no background tasks. The design can also be considered to be reactive.

```
global variable declarations

isr set up
function prototypes
void main (void)
{
    local variable declarations
    while(1);          // task loop
}
ISRs
function definitions
```

Figure 12.3 An Event-Driven Schedule Algorithm

When an interrupting event occurs, flow of control jumps to the associated ISR where the designated task is executed; flow then resumes in the infinite loop. Generally, the event originates from some external source. We will look at an extension to the event-driven approach in which the event derives from a system timer.

The overall behavior of such a system can be difficult to analyze because of the non-deterministic nature of asynchronous interrupts. However, it is rather straightforward to determine the postevent behavior for systems with a single interrupt or the behavior of the highest priority interrupt in systems with more than one interrupt.

Polled and Polled with a Timing Element

The basic polled algorithm is among the simplest and fastest algorithms. The system continually loops, waiting for an event to occur. The difference between the polled algorithm and the event driven is that the polled algorithm is continually testing the value of the polled signal looking for a state change. The interrupt-driven design, on the other hand, does nothing until the event occurs. Only then does it respond. Schematically, the algorithm is given as shown in Figure 12.4.

Such a scheme works well for a single task. It is completely deterministic. The time to respond to the event is computable and bounded. In the worst case, let's assume the event occurs immediately after the test instruction. Under such a circumstance, the response time is the length of the loop. Polled with a timing event is a simple extension. The scheme uses a timing element to ensure a delay action after a polled event is true. Such a technique deskews the incoming signals.

The polled model is also a special case of the foreground/background model. In contrast to the event-driven schedule, the polled model has no foreground tasks. The design implements a reactive system.

```
global variable declarations
function prototypes
void main (void)
{
    local variable declarations
    while(1)            // task loop
    {       // test state of each signal in polled set
            if then construct
                or
            switch statement
    }
}
function definitions
```

Figure 12.4 A Polling-Based Schedule Algorithm

12.3.3 State Based

The next approach implements the flow of control through the task set as a finite automaton or state machine. The two basic implementations of the finite-state machine (FSM), Mealy and Moore, are distinguished by the implementation of the output function: in Mealy the output is a function of the current state and the input, and in Moore the output is a function of the current state only. The basic machine can be expressed as illustrated in Figure 12.5.

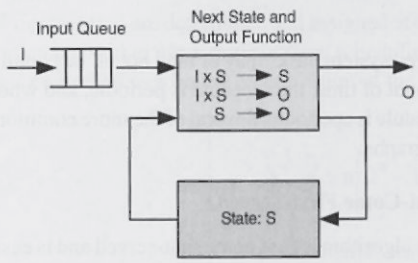


Figure 12.5 A Basic State Machine Model

The state machine can easily be implemented as either a set of case statements, as an if-then, or if-then-else construct.

Some of the limitations of such an approach begin with the theoretical limit on the computational power of the finite-state machine. Using states is not efficient, and the state space explosion for large problems makes the approach impractical for systems with large numbers of inputs. There is a rich set of variations on the basic FSM, however, some of which address the various limitations of the basic implementation. A state-based design is reactive in nature.

12.3.4 Synchronous Interrupt Event Driven

The next level of sophistication entails constraining the asynchronous event used in the opening algorithm to one that is synchronous, based on a timer. Such a system continually loops until interrupted by a *timing* signal (which is typically internally generated). The timing/interrupt event triggers a context switch to an ISR that manages it. A schedule based on

timing

a periodic event is defined as fixed rate. In contrast, an aperiodic schedule is defined as sporadic. Such a synchronous interrupt-based scheme can work with multiple tasks and is the basis for *time-sharing systems*. The design is an example of a time-based system, although it is reacting to a special interrupt.

time-sharing systems

12.3.5 Combined Interrupt Event Driven

A simple variation on the two interrupt event-driven designs is to permit both synchronous and asynchronous interrupts. In such a system, priority is used to select among tasks that are ready when the timing interrupt occurs. If multiple tasks are permitted to have the same priority, then selection from among ready tasks proceeds in a round robin fashion. Naturally, higher priority tasks will be given preference at any time.

12.3.6 Foreground–Background

foreground–background
foreground
background

A system utilizing a *foreground–background* flow of control strategy implements a combination of interrupt and noninterrupt-driven tasks. The former are designated the *foreground* tasks and the latter the *background* tasks. The background tasks can be interrupted at any time by any of the foreground tasks and are thus operating at the lowest priority. The interrupt-driven processes implement the real-time aspects of the application; the interrupt events may be either synchronous or asynchronous. All of the previous algorithms are special cases of foreground/background designs in which either the foreground (polled systems) or the background (interrupt based) component is missing.

12.3.7 Time-Shared Systems

In a time-shared system, tasks may or may not all be equally important. When all are given the same amount of time, the schedule is periodic, and when the allocation is based on priority, the schedule is aperiodic. Several of the more common algorithms are examined in the ensuing paragraphs.

12.3.7.1 First-Come First-Served

A very simple algorithm is first-come first-served and is easily managed with a FIFO queue. When a process enters the ready queue, the task control block is linked to the tail of the queue. When the CPU becomes free, it is allocated to the process at the head of the queue. The currently running process is removed from the queue. Such an approach is nonpreemptive and can be troublesome in a system with real-time constraints.

12.3.7.2 Shortest Job First

The shortest job first schedule assumes that the CPU is used in bursts of activity. Each task has associated with it an estimate of how much time the job will need when next given the CPU. The estimate is based on measured lengths of previous CPU usage. The algorithm can be either preemptive or nonpreemptive. With a preemptive schedule, the currently running process can be interrupted by one with a shorter remaining time to completion.

12.3.7.3 Round Robin

The round robin algorithm is designed especially for time-shared systems. It is similar to first-come first-served, with preemption added to switch between processes. A small unit of

time quantum, slice

time called *time quantum* or *slice* is defined, and the ready queue is treated as a circular queue. The scheduler walks the queue, allocating the CPU to each process for one time slice. If a process completes in less than its allocated time, it releases the CPU; otherwise, the process is interrupted when time expires and it's put at the end of queue. New processes are added to the tail of the queue. Observe that if the time slice is increased to infinity, round robin becomes a first-come first-served scheduler.

12.3.8 Priority Schedule

Shortest job first is a special case of the more general priority scheduling class of algorithms. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priority jobs are scheduled first-come first-served or in round robin fashion. The major problem with a priority schedule is the potential for indefinite blocking or starving—priority inversion. The algorithms can be either preemptive or nonpreemptive.

12.3.8.1 Rate-Monotonic

rate-monotonic

With a preemptive schedule, the currently running process can be interrupted by any other task with a higher priority. A special class of priority-driven algorithms called *rate-monotonic* was initially developed in 1973 and has been updated over the years. In the basic algorithm, priority is assigned based on execution period; the shorter the period, the higher the priority.

static, fixed

Priorities that are determined and assigned at design time and then remain fixed during execution are said to use a *static* or *fixed* scheduling policy. The ability to schedule a set of tasks is computed as a bound on utilization of the CPU as shown in Eq. 12.1.

$$\sum_{i=0}^{n-1} \frac{e_i}{p_i} \le n\left(2^{\frac{1}{n}} - 1\right) \tag{12.1}$$

e = Execution time of the task
p = Period of the task

This approach makes the following assumptions.

- The deadline for each task is equal to its period.
- Any task can be preempted at any time.

The expression on the right-hand side gives a bound on CPU utilization; the bound is extreme, that is, worst case. If it cannot be met, a more detailed analysis must be performed to prove whether or not the task can be scheduled. The above equation sets a CPU utilization bound at 69%. Practically, the bound could be relaxed to around 88%, and the tasks can still be scheduled.

The basic algorithm given above simplifies system analysis. Scheduling is static, and the worst case occurs when all the jobs must be started simultaneously. Formal analysis that is beyond the scope of this text leads us to the rate-monotonic schedule also known as the *critical zone theorem.*

critical zone theorem

> *Critical Zone Theorem*
> If the computed utilization is less than the utilization bound, then the system is guaranteed to meet all task deadlines in all task orderings.

It can be shown that rate-monotonic systems are the optimal fixed rate scheduling method. If a rate-monotonic schedule cannot be found, then no other fixed rate scheme will work. The algorithm is defined as *stable,* which means that as additional, lower priority tasks are added to the system, the higher priority tasks can still meet their deadlines even if lower priority tasks fail to do so. The initial algorithm bases assurance upon the assumption that there is no task blocking. The basic algorithm can be modified to include blocking as illustrated in Eq. 12.2.

stable

$$\sum_{i=0}^{n-1} \frac{e_i}{p_i} + \max\left(\frac{b_0}{p_0},\ldots,\frac{b_{n-1}}{p_{n-1}}\right) \le n\left(2^{\frac{1}{n}} - 1\right) \tag{12.2}$$

The terms b_i give the maximum time task i can be blocked by a lower priority task

With a nonpreemptive schedule, a currently arriving higher priority process is placed at the head of the ready queue.

12.3.8.2 Earliest Deadline

earliest deadline

A dynamic variation on the rate-monotonic algorithm is called *earliest deadline*. The earliest deadline schedule uses a dynamic algorithm with priority assigned based on the task with the closest deadline. The schedule must be established and modified during runtime, for only then can the deadline(s) be assessed.

A set of tasks is considered schedulable if the sum of the task loading is less than 100%. It is considered *optimal* in the sense that if a task can be scheduled by other algorithms, then it can be scheduled by the earliest deadline.

The algorithm is not considered stable. If the runtime task load rises above 100%, some task may miss its deadline. Generally, it is not possible to predict which task will fail. This uncertainty adds greater runtime complexity. The scheduler must continually determine which task to execute next whenever such decisions must be made. Such analytical methods are more complex than fixed priority cases.

12.3.8.3 Least Laxity

least laxity

The *least laxity* algorithm is similar to the earliest deadline with slightly tighter constraints. In addition to the deadline, the time to execute the task is considered. Task priority is based on the following relationship. It should be clear that a task with negative laxity cannot meet its deadline.

$$\text{laxity} = \text{deadline} - \text{execution time} \tag{12.3}$$

The schedule is then based on the metric using ascending laxity. On paper it is a rather straightforward concept. However, it means that one must know the exact value of the exe-

cution time, or at least an upper bound on it. Furthermore, the values must be updated with each system change.

The least laxity algorithm can be utilized in systems with a mixture of hard and soft deadlines. Hard real-time tasks can be given priority over those with less rigid constraints. However, it has weaknesses similar to those found with the earliest deadline algorithm; that is, it is not stable. In addition, it has a greater runtime burden than the fixed schedule schemes. The algorithm tends to devote CPU cycles to tasks that are clearly going to be late and thereby causes more tasks to miss deadlines.

12.3.8.4 Maximum Urgency

maximum-urgency-first

criticality

critical, noncritical

The *maximum-urgency-first* algorithm includes features of both the rate-monotonic and the least laxity algorithms. As a first cut, it assigns priority according to the task's period, as is done with the rate-monotonic algorithm. Next, a binary *criticality* task parameter is added. The criticality parameter is used to decompose the tasks into two sets: *critical* and *noncritical*. Then the least laxity algorithm is applied to those in the critical set. The criticality parameter and the priority assignment are assessed at runtime.

If no critical tasks are waiting, then tasks from the noncritical set are scheduled. Because the critical set is based on the rate-monotonic algorithm, the schedule can be structured so that no critical task fails to meet its deadline.

The major advantage of the algorithm is the simplicity of the static priority component and reduced runtime burden compared with full least laxity. The algorithm, however, lacks some flexibility. The rate-monotonic component assumes unconstrained preemption. Typically, short deviations are well tolerated; longer deviations can lead to missed deadlines.

Maximum-urgency-first is best applied to tasks that are well understood and for which blocking constraints are easy to determine. The dynamic scheduling contribution from least laxity potentially can compensate by elevating a task's priority. The algorithm has some of the runtime complexity of pure least laxity and is best applied to tasks that can vary in their ability to miss deadlines. It can be thought of primarily as a rate-monotonic algorithm with some runtime checking to ensure that deadlines can be met.

12.4 REAL-TIME SCHEDULING CONSIDERATIONS

A real-time system may be hard or soft real-time, and the task scheduling may be static or dynamic. For a dynamic hard real-time schedule, the process is submitted along with a statement of the time required to compute and to do I/O. If, following assessment of the task's requirements, the scheduler accepts the task, it guarantees that the task will complete on time. Otherwise, it rejects the task as nonschedulable. Such a guarantee calls for *resource reservation* and requires the scheduler to know exactly how long each operating system function takes along with a completion time guarantee. Such a restriction is impossible for systems with secondary storage or using virtual memory algorithms.

resource reservation

A soft real-time schedule is less restrictive. Such a schedule does require that critical processes have priority over the less critical. Implementing a soft real-time system requires careful design of the scheduler and other related aspects of the operating system. There is a further requirement for priority scheduling. Real-time processes must have the highest priority, and that priority must not degrade over time. Such a constraint is relatively easy to ensure. Furthermore, the dispatch latency must be small; thus, system calls must be preemptable.

Such a requirement can be accomplished in several ways. One approach is to insert preemption points where the system can check to see if a high-priority process needs to be run. Alternatively, the entire kernel can be made preemptable. In such a case, all kernel data structures must be protected, and one must have synchronization methods.

conflict phase, dispatch phase

The preemption process has two components: a *conflict phase* and a *dispatch phase*. During the conflict phase, preemption of any process running in the kernel is permitted. The lower priority process must release needed resources. The next step is a context switch to the high-priority process. In the dispatch phase, the process moves from the ready state to the run state.

12.5 ALGORITHM EVALUATION

With the plethora of algorithms and each having its own parameters, selecting the proper and appropriate one can be difficult. To begin the evaluation, one must first establish assessment criteria. For example, CPU utilization, response time, or throughput may be the most critical factors in a design. Next, the candidate algorithms must be evaluated against the selection criteria. Once again, there are a variety of methods.

12.5.1 Deterministic Modeling

analytic evaluation

deterministic modeling

A major class of methods is called *analytic evaluation*. The approach uses the candidate algorithm and a representative system workload to produce a formula or number from which to evaluate the algorithm. One such method is called *deterministic modeling*. To see how this works, consider the following processes and workloads.

Process	Burst Time
P1	10
P2	20
P3	3
P4	7
P5	12

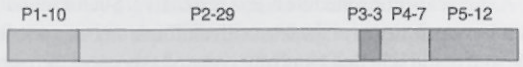
Figures 12.6a–c illustrate the results of evaluating the following scheduling algorithms against the example workload.

- First-come first-served
- Shortest job first
- Round robin

first-come first-served

Starting with the *first-come first-served*, each algorithm will be evaluated with the goal of achieving the shortest average wait time.

First-Come First-Served



Process	Waiting Time
P1	0
P2	10
P3	32
P4	42
P5	49

Average 28 time units

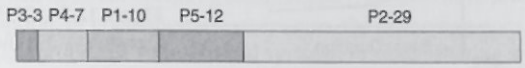
Figure 12.6a The First-Come First-Served Algorithm

It is assumed that the jobs arrive into the system in the order shown. With this algorithm, the average wait time is computed to be 28 time units.

shortest job first

Next is the *shortest job first* schedule.

Shortest Job First



Process	Waiting Time
P3	0
P4	3
P1	10
P5	20
P2	32

Average 13 time units

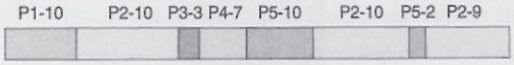
Figure 12.6b The Shortest Job First Algorithm

Now, the average wait is 13 time units. The algorithm achieves a two to one improvement over the FIFO schedule.

round robin

For, the *round robin* algorithm, the time slice is set to 10 time units. Under such a constraint, jobs P1, P3, and P4 will complete in their allotted time. P2 and P5 will have to be preempted and returned to the queue.

Round Robin



Process	Waiting Time
P1	0
P2	32
P3	20
P4	23
P5	40

Average 23 time units

Figure 12.6c The Round Robin Algorithm

shortest job first

Now the average wait is 23 time units. In the above example, clearly the *shortest job first* algorithm should be the choice since it performs the best against the specified metric.

As can be seen, deterministic modeling is simple and fast, but it does require exact knowledge of the process times, which often can be difficult to establish. One obvious solution is to measure the process times over repeated executions. Such data collection can be done more easily in the embedded world than in the applications world because one generally knows the task mix in advance.

5.2 Queuing Models

If the system being designed is one in which the processes can vary from day to day, there may be no static set of processes and times that can be used in a deterministic model.

Statistical studies have shown that task execution generally consists of a cycle of CPU execution followed by I/O activity. The CPU and I/O bursts alternate until the job is finished. The frequency of the bursts tends to be fairly predictable and is typically independent of machine or process. As a first-order approximation, such behavior can be modeled as the exponential graph given in Figure 12.7. One can measure or compute the distribution of

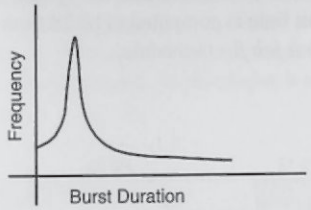


Figure 12.7 CPU or I/O Burst Duration vs. Frequency

CPU and I/O bursts over a collection of tasks and determine a similar distribution for process arrival times. Based on these two distributions, for most algorithms, it is possible to compute average throughput, utilization, waiting times, and so on.

The computer can be modeled as a collection or network of servers, with each server having an associated queue. Knowing the arrival and service rates, one can compute utilization, the average queue length–n, and the average wait time–w. The average arrival time is specified as λ. Thus, if the system is in steady state, the number of processes leaving a queue is equal to the number of processes arriving, and one can write,

$$\boxed{n = \lambda \times W} \tag{12.4}$$

Little's Formula

The expression relating the three variables is known as *Little's formula*. The approach is useful because it is valid for any scheduling algorithm. Knowing any two variables, one can compute the third. Though useful for comparing algorithms, it has limitations. The mathematics of complex algorithms and distributions is difficult to work with. The arrival and service distributions are complex, and the queuing models are only an approximation of the real system.

.5.3 Simulation

To produce a more accurate evaluation of a scheduling algorithm, one can use simulations. Such an approach requires models of the computer system and the processes as well as appropriate data to drive the simulation. Often such data is collected from a trace of actual processes by recording the actual events on a real system. Simulation can be expensive, but it is growing in popularity and is becoming an increasingly powerful and effective tool.

.5.4 Implementation

As another alternative, one can simply build and test the system. Certainly, this is the most accurate method. Once again, the difficulty is the cost.

.6 TASKS, THREADS, AND COMMUNICATION

.6.1 Getting Started

exchanging data synchronizing, sharing resources

A multitasking/multithreading system supports multiple tasks, and those tasks will have one or more threads. Important jobs in any multitasking system include *exchanging data* between tasks and between threads, *synchronizing* tasks and threads, and *sharing resources*.

In the not too recent past, such activities were limited primarily to tasks or threads within a single microprocessor. Today, one finds a growing use of FPGA-based designs utilizing devices that support the inclusion of multiple microprocessor cores within a single-gate array. Consequently, it is not uncommon for communication, synchronization, and sharing to involve tasks on multiple processors. We will find that certain assumptions can

be made when tasks are localized that cannot be made when working with multiple distributed processors or other centers of computation.

12.6.2 Intertask/Interthread Communication

When tasks are operating independently, systems have few if any conflicts, chances for corruption, or contentions. Real systems, the interesting ones, must deal with the challenge of such problems. In real-world systems, resource sharing and intertask synchronization and communication must take place in a robust, safe, and reliable manner. Interaction between tasks may be direct or indirect and must be synchronized and coordinated. We want to prevent race conditions—conditions under which the outcome of a computation depends on the order in which tasks execute. Such an exchange is illustrated in Figure 12.8.

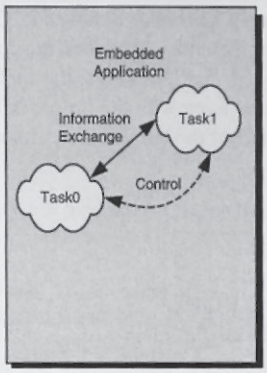


Figure 12.8 Intertask Communication

We see, then, that interaction and interchange among tasks requires three basic components: the information that is to be interchanged, the places where the information can be found, and where it is ultimately to be put, coupled with the conventions that govern the interaction and interchange. These requirements are captured in the following model of interprocess communication and synchronization.

information
- The *information*—the data or signals being moved

place, places
- The *place* or *places* from which the information is moved to or from

control, synchronization
- The *control* and *synchronization* of actions and the movement of the information

places

In such a model, the *places*—that is, the source and destination(s) for the exchange—are identified variously by named variables or by pointer variables holding memory addresses. Control and coordination comprises a number of different techniques ranging from flags or status bits to interrupts or managed access into critical areas under the control of semaphores or monitors. Information is moved either through *shared variables,* or *messages* on busses internal to the microprocessor that (except in rare circumstances) were of little immediate concern to us.

shared variables, messages

Let's begin our study of intertask communication and synchronization by looking at the shared information component. Such sharing can occur in a variety of ways. In subsequent chapters, we will extend the model to include centers of control outside of the core microprocessor.

12.6.3 Shared Variables

Such sharing can occur in a variety of ways. We will begin with the simplest model: shared global variables.

META-1026, p. 17
Meta v. Resonant
IPR2025-00823

12.6.3.1 Global Variables

global variables

One fundamental solution for exchanging data among tasks is a shared memory environment. In such an environment, *global variables* can be a very effective mechanism for sharing information. Global variables have the obvious problems that arise when two or more tasks require the ability to read a piece of global data and potentially modify its value. The major advantage of globals is that they do not have to be copied to the stack during a context switch. By obviating the need for such copying, critical time in hard real-time systems can be saved. Properly managed, global variables can be an effective tool.

12.6.3.2 Shared Buffer

shared buffer
producer
consumer

A *shared buffer* is an exchange technique in which two processes share a common set of memory locations as seen in the data flow diagram in Figure 12.9. A *producer* of the data puts it into the buffer, and a *consumer* removes it. Once again, there are several obvious problems. If one process is faster than the other, the potential for overrun or underrun arises. Clearly, identifying the proper buffer size (for the application) and access protocol is critical to avoiding such problems. Even with the proper buffer size, the producer and consumer must always check the state of the buffer before inserting or removing an item.

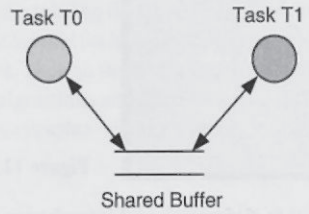


Figure 12.9 Intertask Communication Using a Shared Buffer

Good design practice recommends adding methods of the form

```
bool isFull() or bool isEmpty()
```

to the public interface of the container. Such methods should always be invoked prior to a read from or write to the buffer.

12.6.3.3 Shared Double Buffer—Ping-Pong Buffer

shared double buffer
ping-pong buffer

The *shared double buffer* model permits two tasks to share two (or more) common sets of memory locations. Shown in the data and control flow diagram in Figure 12.10, the configuration is also called a *ping-pong buffer*.

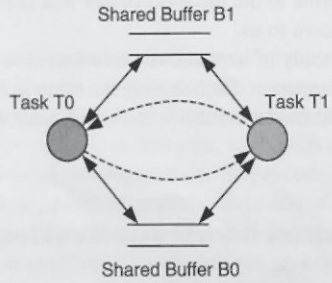


Figure 12.10 Intertask Communication Using a Shared Double Buffer

Several control schemes can be used with a ping-pong buffer. One implementation begins with both buffers being empty. T0 is designated as the producer and T1 as the consumer. During operation, task T0 will write to buffer B0 until it is full. In the meantime, T1 is blocked because there is no data available. Once B0 is filled, T0 will signal T1 and switch to writing to buffer B1.

T1 can now begin reading the data from B0. When T1 has removed all the data from B0, it signals T0 that the buffer is empty. If T0 has filled B1, T1 can begin reading from that buffer; otherwise it waits. Similarly, if T0 finishes writing to B1 before T1 has emptied B0, then it must block. The operation of the buffer scheme is illustrated in the two skeletal code fragments in Figure 12.11. Such an approach can be a very effective "buffer" between processes that are running at different rates. One buffer is being filled while the other is being emptied. Improved robustness requires that the consumer block on a lack of data and the producer must avoid overrunning the buffer; thus, it blocks on a full buffer.

```
Task T0
while(1)
    ...
    if (B0 == EMPTY)
        repeat
            produce item in nextB0
        until (B0 == FULL)
        signal (T1, FULLB0)
    endif
    if (B1 == EMPTY)
        repeat
            produce item in nextB1
        until (B1 == FULL)
        signal (T1, FULLB1)
    endif
    ...
end while
```

```
Task T1
while(1)
    ...
    if (B0 == FULL)
        repeat
            consume item in nextB0
        until (B0 == EMPTY)
        signal (T0, EMPTYB0)
    endif
    if (B1 == FULL)
        repeat
            consume item in nextB1
        until (B1 == EMPTY)
        signal (T0, EMPTYB1)
    endif
    ...
end while
```

Figure 12.11 Two Tasks Exchanging Information Using a Shared Buffer

A second variant on the ping-pong buffer utilizes more than two buffers. Consider that we have two tasks, T0 and T1; the first task can produce data at a rate of 4 MHz, but the second can only consume at 1 MHz. To further complicate the problem, let's also assume that the buffer can only be written to at a 1 MHz rate. An implementation to solve the problem is given in the data and control flow diagram in Figure 12.12.

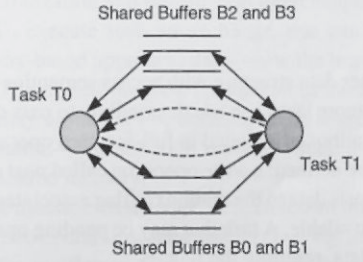


Figure 12.12 Information Sharing Between Tasks Executing at Different Speeds

The execution of the synchronization scheme is given in the pseudo-code fragments in Figure 12.13. Each buffer is written to at the 1-MHz rate. The buffers are filled in bursts in T0 and read at a more uniform rate in T1.

```
Task T0
while(1)
    ...
    if (B3 == EMPTY)
        repeat
            produce item in nextB0
            produce item in nextB1
            produce item in nextB2
            produce item in nextB3
        until (B3 == FULL)
        signal (T1, FULLB3)
    endif
    ...
end while
```

```
Task T1
while(1)
    ...
    if (B3 == FULL)
        repeat
            consume item in nextB0
            consume item in nextB1
            consume item in nextB2
            consume item in nextB3
        until (B3 == EMPTY)
        signal (T0, EMPTYB3)
    endif
    ...
end while
```

Figure 12.13 Information Sharing Between Tasks Executing at Different Speeds

12.6.3.4 Ring Buffer

ring buffer

A *ring buffer* scheme uses a FIFO structure as illustrated in the accompanying schematic representation in Figure 12.14. The structure permits simultaneous input and output using head and tail pointers. Task T0, the producer, adds data to the buffer, and task T1, the consumer, removes it. As with the other buffers, one must take precautions to properly manage overflow and underflow.

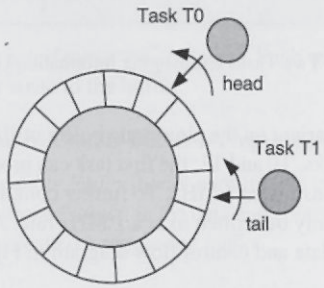


Figure 12.14 Information Sharing Using a Ring Buffer

12.6.3.5 Mailbox

mailbox

A *mailbox* is another data structure with access semantics that are similar to those used for the queue. Two or more tasks can use the mailbox to pass data or for synchronization. Generally, one finds mailboxes included in full-featured operating systems. Two operations on the data structure are defined: a write operation called *post* and a read operation called *pend*.

post, pend

When a task posts data to the mailbox, a flag associated with the mailbox is raised, indicating that data is available. A task that may be pending or waiting on that flag is alerted and can then read the data, resetting the flag.

The pend and post operations present the following public interface:

```
post (mailbox, data)        // post to mailbox
pend (mailbox, data)        // pend on mailbox
```

At first blush, the pend operation may appear to be the same as a poll because a poll task continually interrogates the polled variable (occupying the CPU) looking for a change in state of the signal. In contrast, however, the pending task is suspended (giving up the CPU), while there is no data available only to be awakened when data becomes available. Thus, in the case of a polling operation, the CPU is devoted to testing the state of the poll signal, whereas the pend operation frees the CPU to another task. A variety of things can be passed through a mailbox, a single bit or flag, a single data word, a pointer to a data buffer, or a more elaborate message.

The data and control flow diagram is given in Figure 12.15.

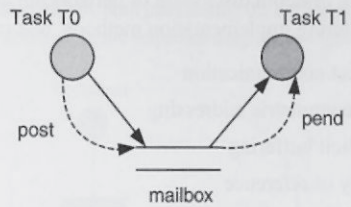


Figure 12.15 Information Sharing Using Messages and a Mailbox

One straightforward implementation of the mailbox data type utilizes a queue as the underlying container. In the basic implementation, the queue is of length one and thus, the post operation fills the mailbox precluding further posts until a pend operation takes place to empty the mailbox. If several tasks are pending on a flag, the enabled task resets the flag. Such a scheme blocks multiple accesses to the resource from a single flag. Other implementations extend the queue length, thereby supporting a queue of pending elements rather than a single entry. Such a scheme may be useful when there are multiple independent copies of a critical resource. Another variation on this latter design utilizes a priority queue and thence permits a priority to be assigned to each message. The associated pend operation will always read the highest priority message first.

12.6.4 Messages

The methods for intertask communication discussed up to this point have relied on a mutually agreed upon memory location to at least begin the exchange. Today's embedded applications are becoming increasingly distributed. With such an expansion, the need for synchronization and information interchange remains and, to some extent, increases.

To execute such an exchange, one can build on the concept of mailboxes. Using a mailbox-based approach, data—now the message—is sent to a named mailbox or destination. The named mailbox now becomes the address of the message destination. The message may or may not be buffered at the source of the message—a source mailbox or at the destination—a destination mailbox. Such a scheme, however, is not mutually exclusive with shared memory.

interprocess communication
send, receive, pend, post

A message-based approach, called an *interprocess communication* facility (IPC), supports two operations, *send* and *receive*. These are analogous to the *pend* and *post* operations used for mailboxes. Continuing the analogy, messages may be of fixed or variable size. If

META-1026, p. 21
Meta v. Resonant
IPR2025-00823

tasks T0 and T1 wish to use messages to exchange information, they must first establish a communication link and then proceed to send and receive the messages.

As noted earlier, with the increasing use of multiprocessor core FPGAs, the communication link can be between processors within the same gate array as well as between physically and geographically separated microprocessors.

As one begins to think about message exchange, several questions immediately arise,

- How is the link established?
- Can the link be associated with multiple tasks?
- How many links are there between a pair of tasks?
- What is the link capacity, and are there buffers?
- What is the message size?
- Are links unidirectional or bidirectional?

We will look at several of these questions but defer the last two to a later chapter in which we present a more in-depth discussion of networking and remote systems.

When considering implementation methods, one may choose

- Direct/indirect communication
- Symmetric/asymmetric addressing
- Auto or explicit buffering
- Send by copy or reference
- Fixed or variable message size

12.6.4.1 Communication

directly, indirectly

A message can be moved from one place to another, either *directly* or *indirectly,* via some intermediate point or points. Each way has advantages and disadvantages.

DIRECT

When using a direct communication scheme, each process must explicitly name the sender/receiver of the message. Messages are logically of the form

```
send (T1, message)        // send message to task T1
receive (T0, message)     // receive message from task T0
```

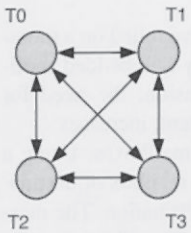


Figure 12.16 Four Fully Interconnected Tasks

The link is automatically established between every pair of processes or threads within a process. For a system with four different processes, the configuration in Figure 12.16 gives full, bidirectional interconnection among all of the processes. Several important points need to be considered with such an implementation:

- The individual tasks may or may not be physically collocated. On one extreme, they may be within the same FPGA. On the other, they could be in several different countries.
- Full interconnectivity is not efficient for larger numbers of tasks. A hierarchical scheme in which a smaller subset of the tasks is so interconnected may be more feasible to implement and manage. Consider the Internet as a good model.

Using a direct communication scheme, each task only needs to know each other's identity; that is, the link is associated with only two processes. The link may be unidirectional or bidirectional.

META-1026, p. 22
Meta v. Resonant
IPR2025-00823

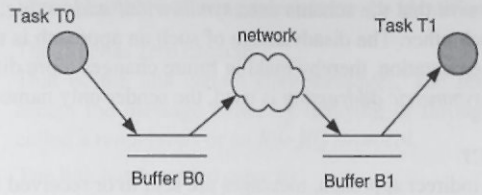


Figure 12.17 Information Exchange Between Two Tasks over a Network

The exchange can be expressed in a modified data flow diagram in Figure 12.17. Note that a buffer is associated with each process, although this may not be the case in all implementations. More specifically, the buffer will probably be attached with an I/O task.

EXAMPLE 12.0

Consider the skeletal structure between two tasks—a producer task, T0 and consumer task, T1. Task T0 produces the data and stores it in a buffer it shares with the send task. The send task takes the data from the buffer and formats it into a message that it sends as the payload in a message to task T1, the consumer task. T1 then reverses the process.

The activities by both tasks during the exchange are first expressed in the activity diagram in Figure 12.18 and then in the sequence diagram in Figure 12.19.

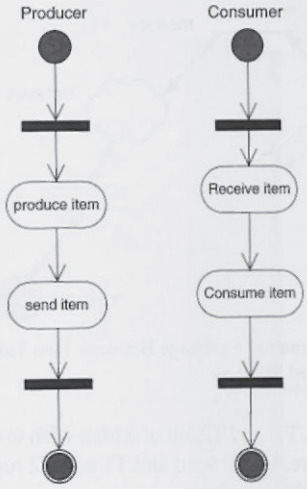


Figure 12.18 Activity Diagram Illustrating a Producer–Consumer Exchange

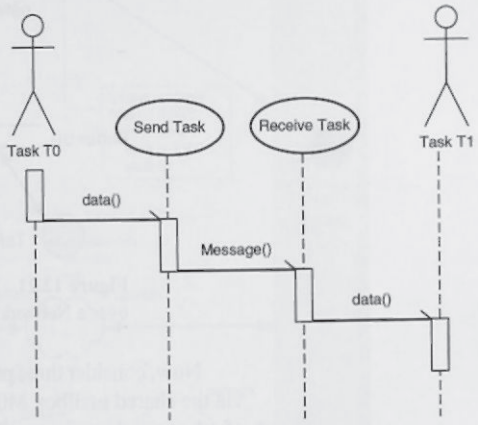


Figure 12.19 Sequence Diagram Illustrating a Producer–Consumer Exchange

Finally, the code fragment shown in Figure 12.20 reflects the operation of the two tasks.

```
while(1)                          while(1)
    ...                               ...
    produce item in nextB0            receive(T0, nextB1)
    ...                               ...
    send (T1, nextB0)                 consume item in nextB1
...                               ...
end while                         end while
```

Figure 12.20 Code Fragment Illustrating a Producer–Consumer Exchange

symmetrical addressing

Observe that the scheme uses *symmetrical addressing*; the sender and receiver must name each other. The disadvantage of such an approach is that it ties the process name to the implementation, thereby making future changes more difficult.

asymmetric addressing

If *asymmetric addressing* is used, the sender only names the recipient.

INDIRECT

With an indirect approach, messages are sent to or received from a shared variable, generally in the form of a mailbox. Thus,

```
send (M0, message)        // send message to mailbox M0
receive (M0, message)     // receive message from mailboxM0
```

The link is established only if the tasks/threads have a shared mailbox or similar container. The link may be associated with multiple processes, and there may be multiple links between processes. As with the direct scheme, the link may be unidirectional or bidirectional. The modified data flow diagram takes the form shown in Figure 12.21, in which two tasks are illustrated. The interconnecting links are shown as bidirectional.

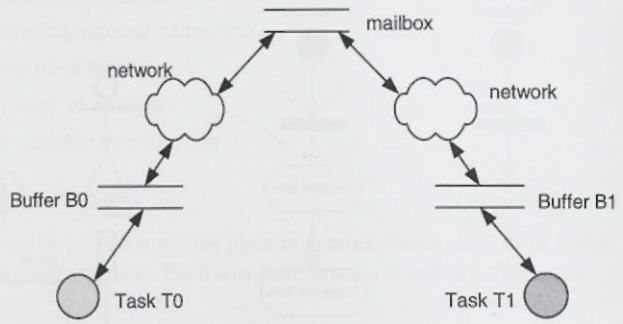


Figure 12.21 Indirect Information Exchange Between Two Tasks over a Network Using a Shared Mailbox

Now, consider three processes: T0, T1, and T2, all of which wish to exchange messages via the shared mailbox M0. Furthermore, let T0 send and T1 and T2 receive. The question of who gets the message, T1 or T2, arises.

One possible solution is to associate the link with at most two processes. Thus, only one process is allowed to receive at a time. As an alternative approach, the system could select a receiver. A third approach can be based on the owner of the mailbox.

If a task owns the mailbox, one can easily distinguish between the owner, who can only receive (there is no reason to send a message to ourselves other than as a built-in test), and the user, who can only send. Since each mailbox has a unique owner, there is no ambiguity. If the system owns the mailbox, then it exists independent of any process or thread.

12.6.4.2 Buffering

A buffer or buffers may be associated with the link. Error management aside, for the moment, buffering establishes the number of messages that can be safely sent out onto the link with the assurance that they will be received properly at the destination. If messages are sent too quickly, the receiver may not have sufficient time to accept and process one message before the next one arrives.

Three possible buffering schemes can be identified.

- The link has zero capacity.
  That is, the link cannot store messages. The sender must wait for the receiver to accept the message either by delaying or through a handshake. Such a scheme is called a *rendezvous* or an *Idle RQ protocol*.

rendezvous
Idle RQ protocol

- The link has bounded capacity.
  Associated with the link is a message queue of length n. If there is space remaining when the sender wishes to transmit, a message can be placed into the queue and the sender can continue. Otherwise the sender must wait for space.
- If the link has unbounded capacity, it can be viewed as having infinite length.
  The sender can post a message and continue. There is no wait. It is important to recognize that the criterion here is that the sender does not have to wait. If the receiver can remove the incoming data quickly enough, a buffer size of one will suffice and can still be called unbounded. Such a scheme is called a *Continuous RQ protocol*.

Continuous RQ protocol

All of the approaches to intertask communication that we have discussed are captured in Figure 12.22.

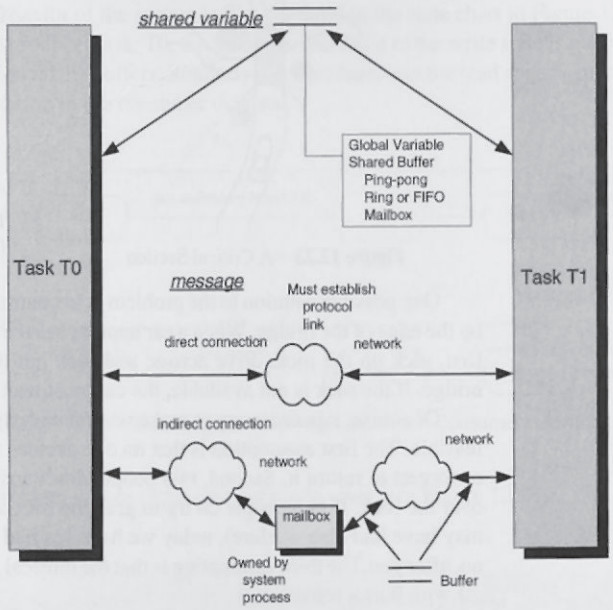


Figure 12.22 Alternative Approaches for Intertask Communication

12.7 TASK COOPERATION, SYNCHRONIZATION, AND SHARING

In addition to sharing information, the tasks in a multitasking system or the processors in a multiprocessor system are often charged with cooperating/synchronizing with each other as they execute the application. Cooperating tasks (and threads) or processors can affect or be affected by other tasks (and threads) or processors. They may directly share a logical address space (both code *and* data) or be allowed to share data only through any of the various shared variable models that have been discussed. Such concurrent access to common

data can result in data inconsistency, aberrant or unexpected system behavior, and potentially complete system failure.

Critical Sections and Synchronization

Northern Scotland is beautiful, rugged, and lightly populated. There are few roads, with little traffic. Many of the roads are narrow, bucolic, single-lane driving challenges populated with passing places and sometimes even narrow bridges as seen in the accompanying simple drawing in Figure 12.23. As the two cars arrive, the bridge clearly presents a problem since it is only wide enough for a single car to cross at any time. Not having both vehicles simultaneously occupying this critical section of the road is most certainly beneficial to all concerned.

If each car is modeled as a process and the bridge as a shared resource, the problem is expressed using the data flow diagram in Figure 12.24.

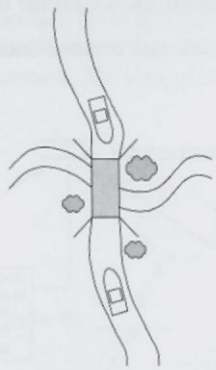


Figure 12.23 A Critical Section

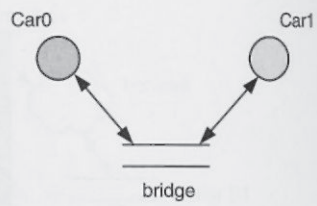


Figure 12.24 A Shared Variable Critical Section

One possible solution to the problem is to control access to the bridge by placing a rock on the edge of the bridge. When a car approaches the bridge and wants to cross, it must stop first, pick up the rock, drive across, and then put the rock back on the other side of the bridge. If the rock is not available, the car must wait.

Of course, it is necessary to make several underlying assumptions for the solution to be feasible. The first assumption is that no one decides to see how far they can throw the rock or forgets to return it. Second, two people don't arrive simultaneously and decide to fight over the rock. If two people do try to grab the rock at the same time (in the olden days we may have had clan warfare), today we have learned to play nice and share—after you; oh no, after you.The third assumption is that the musical group from England doesn't go rolling off with it as a souvenir.

The data flow diagram is extended and illustrated in Figure 12.25 in order to add control, and the design begins to look a bit like a mailbox.

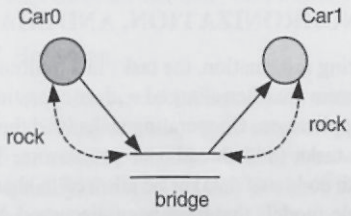


Figure 12.25 Adding Control to Manage a Critical Section

Let's examine how concurrent access to a shared resource can be manifest in a design. Consider the problem that subsequently arises in the accompanying pseudo-code and code fragments. Implemented is a simple data transfer between two tasks, one a producer and the other a consumer, via a shared buffer. The buffer has a limited capacity of *n* items. The transfer must be managed to ensure that the producer does not try to put data into the buffer when it is full and the consumer must not try to take data out when the buffer is empty. A variable *count* provides a measure of the number of items in the buffer. It is incremented when an item is added and decremented when one is removed. The data flow diagram for the shared buffer is given in Figure 12.26.

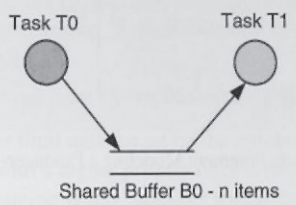


Figure 12.26 Producer–Consumer Exchange Through a Shared n Item Buffer

not full
not empty

The behavior of the system is first captured in the state chart in Figure 12.27. Observe that for the producer task, T0, the transition from idle to the write state is guarded by the *not full* condition on the buffers. Similarly, the transition into the read state is guarded by the *not empty* condition in the consumer diagram.

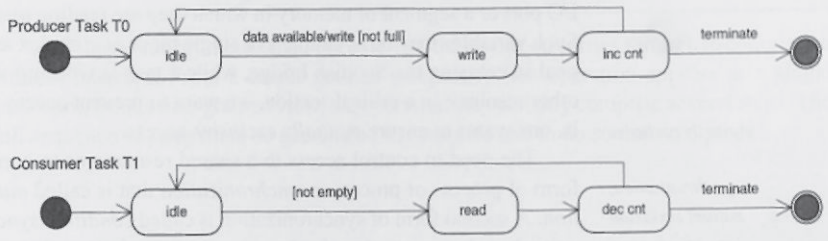


Figure 12.27 State Chart Diagram Modeling a Producer–Consumer Information Exchange

The problem is then expressed in pseudo code (see Figure 12.28).

```
Task T0 - Producer
    while(1)
        If not full
            add item
            increment count
        else
            wait for space
    end while

Task T1 - Consumer
    while(1)
        If not empty
            get item
            decrement count
        else
            wait for item
    end while
```

Figure 12.28 Pseudo-Code Modeling a Producer–Consumer Information Exchange

The C code fragments are given in Figure 12.29.

```
Task T0 - Producer
    int in = 0;
    while(1)
    {
        // produce an item nextT0
        // wait for room
        while (count == MAXSIZE);
        B0[in] = nextT0;
        in = (in + 1) % MAXSIZE;
        count++;
    }
```

```
Task T1 - Consumer
    int out = 0;
    while(1)
    {
        while (count == 0);    // wait for item
        nextT1 = B0[out];
        out = (out + 1) % MAXSIZE;
        count--;
        // consume an item nextT1
    }
```

Figure 12.29 C Code Fragment Modeling a Producer–Consumer Information Exchange

count

As with the attempts at simultaneous access to the bridge, there is a potential problem with simultaneous access to *count*. The value of the variable *count* depends on which task accesses it and in which order. Because the two tasks are running asynchronously, the variable may have any of three different values at any instant in time. Like the bridge, count represents a critical piece of data or *critical section* shared between the two processes, T0 and T1.

critical section

In general, a critical section is a resource that several tasks may be sharing such as an I/O port or a segment of memory in which they are reading *and* writing common variables. Such variables may be as simple as a single bit or as complex as a file or a table. As was the goal in crossing the Scottish bridge, while a task is working with a piece of data or some other resource in a critical section, we want to prevent access by all other processes. That is, one wants to ensure *mutually exclusive* access.

mutually exclusive

synchronization, mutual exclusion synchronization, condition synchronization

The need to control access to a shared resource or to common data gives rise to one form of process or processor *synchronization* that is called *mutual exclusion synchronization*. A second form of synchronization is called *condition synchronization*. For the case of mutual exclusion synchronization, the objective is to make certain that two processes are not in their critical sections at the same time. Condition synchronization, on the other hand, requires that a process delay or block until a specified condition is true (or false).

The need to share and to coordinate access exists only if there is more than one task or processor that wishes to use a nonsharable resource or to modify common data at the same time. This is a key point. If the resource is sharable or if the tasks are executing read only operations, there will be no problem.

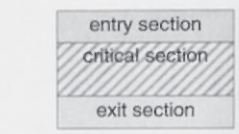


Figure 12.30 An Abstract Model of a Critical Section

As we sought to accomplish with the simple bridge management schemes, the solution to the critical section problem requires a control algorithm or protocol that regulates access to the shared area. At a high level, the protocol should be such that a task wishing to access the critical section should check to see if anyone else is using the variable; if not, announce to all other tasks that it is now going to use the variable, do its work, and then tell everyone when it is finished.

An abstract model of the structure of a task with a critical section can be depicted as shown in Figure 12.30.

entry section

The code relevant to the critical section is enclosed in the three rectangles shown in the figure. The top rectangle, the *entry section*, acts as the gatekeeper controlling access to the

exit section

critical region. The bottom rectangle, the *exit section*, serves to tell the world that the task that had been using the critical variable is now finished.

Any solution to the critical section problem must satisfy the following requirements.

mutual exclusion

- It must ensure *mutual exclusion* in the critical region.
  If a task is in the critical section, no other task may be allowed in.

deadlock

- It must prevent *deadlock*.
  If two or more tasks are trying to enter the critical section, one must succeed.

progress

- It must ensure *progress* through the critical section.
  If no task is in the critical section and some other task wishes to enter, only tasks that are *not* in the exit section rectangle can affect which task enters the critical section next. Furthermore, a task wishing to enter cannot be prohibited from doing so indefinitely.

bounded waiting

- The solution must ensure *bounded waiting*.
  An upper limit must be set on the number of times a lower priority task can be blocked by one with a higher priority once it has made a request to enter.

atomic

Let's examine several possible solutions to the critical section problem. We will begin with a flag-based approach. Prior to doing so, however, we introduce the word *atomic* as a qualifier to an operation.

> *Atomic Operation*
> One that is guaranteed to terminate and is indivisible when applied to either examining a program variable or modifying the state of such a variable.

Indivisible simply means that, once started, the operation carries through to completion without interrupt. From a coarse-grained perspective, the operation appears as a single statement; from a fine-grained view, the operation may actually comprise several steps. The full sequence of steps must be guaranteed to complete and to do so uninterrupted.

7.2 Flags

To protect a critical section, the first goal is to ensure mutually exclusive access. This exclusion can be accomplished using flags embedded in an atomic operation. The method is illustrated using two flags and two processes. Expansion to a greater number of processes follows logically.

Define two processes, T0 and T1. Let them share a critical section. Define two Boolean flags, T0Flag and T1Flag, to mark which process is in the critical section. Finally, define the atomic operation, *await*, which is expressed in pseudo code as shown in Figure 12.31.

await

```
await( condition )
{
        statements
} variable.
```

Figure 12.31 Await Statement Pseudo-Code Model

condition
statements

Condition is a Boolean expression on which a task, thread, or processor waits until it evaluates to true. *Statements* comprise a set of actions that are to be performed when the condition evaluates to true. If the condition evaluates to true, execution proceeds through

await *awaiting* the statements comprising the body of the *await* construct. An important assumption here is that when a process is *awaiting* a condition, other processes have the opportunity to run. Otherwise there is a deadlock.

Using the *await* operation, one can now reexamine the earlier shared buffer problem. The *await* statements are expressed, one for each task, as

```
await(!T1Flag) {T0Flag = true;}
await (!T0Flag){T1Flag = true;}
```

await *count* Next, the *await* statements are used to control access to the critical section—the variable *count*. First, we look at the producer (see Figure 12.32a).

```
Task T0 - Producer
    int in = 0;
    while(1)
    {
        // produce an item nextT0
        while (count == n);                    // wait for room
        B0[in] = nextT0;
        in = (in + 1) % n;
        await( !T1Flag ) {T0Flag = true;}      // entry section
        count++;                               // critical section
        T0Flag = false;                        // exit section
    }
```

Figure 12.32a Managing a Critical Section Using the Await Statement—Producer Side

Then we look at the consumer (Figure 12.32b).

```
Task T1 - Consumer
    int out = 0;
    while(1)
    {
        while (count == 0);                    // wait for item
        nextT1 = B0[out];
        out = (out + 1) % n;
        await( !T0Flag ) {T1Flag = true;}      // entry section
            count--;                           // critical section
        T1Flag = false;                        // exit section
        // consume an item nextT1
    }
```

Figure 12.32b Managing a Critical Section Using the Await Statement—Consumer Side

12.7.3 To

12.7.4

It is rather straightforward to show that this scheme satisfies the first three conditions for solving the critical section problem. Ensuring eventual access is a bit more involved and is contingent on the scheduling policy.

3 Token Passing

Another possible solution to the shared buffer problem is an extension of the rock-passing protocol developed for the Scottish bridge problem. We define a flag or token. To ensure sharing of the data, only one token is issued. The token is continuously passed from task to task; any task wishing to access the critical section can only do so when it has the token as illustrated in the state chart in Figure 12.33. The transition from state A to state B, from which the access to the shared variable occurs, is guarded by the requirement of possessing the token.

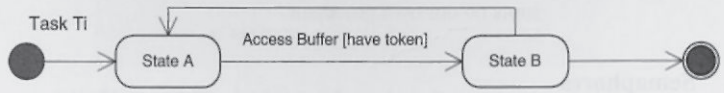


Figure 12.33 State Chart Modeling a Token Passing Protocol as a Solution to the Critical Section Problem

Although there is now controlled access to the critical section, several problems arise immediately:

1. A task or processor that does not want to share holds onto the token forever.
2. The task or processor with the token crashes for an extended time.
3. The token gets lost or corrupted because of noise.
4. The task or process with the token terminates or leaves the system without releasing the token.
5. How does one identify a new task or processor that gets added to the system?

One possible solution to all of these problems is to borrow an idea from our network colleagues. A system-level task, charged with managing the token, is added. The task includes a watchdog timer. Each time the token is released, the timer is reset. If the timer expires, a ping message is sent to all tasks or processors querying for the token. If no one responds, a new token is generated.

Borrowing again from the network people, each time a task or processor enters or leaves the system, it must register with the token management task. Alternatively, the system task could periodically query for new entries into the system.

It is evident that such a protocol satisfies all of the requirements stipulated above and thus does solve the critical section problem. The approach, however, adds a significant intra- and intersystem communication burden as well as extra overhead to each task.

.4 Interrupts

Another approach to solving the buffer problem centers on managing interrupts. Since the problem only arises in a single-processor context when preemption is allowed, preventing preemption solves it. Disallowing all preemption is a bit too extreme in most cases. Taking a more surgical approach offers a more practical path.

Referring back to the earlier figure describing a task with a critical section, we should be able to solve the problem if interrupts are disabled when entering the rectangle labeled *entry* section and reenabled in the section labeled *exit* section.

entry

Using such an approach, one can encounter some of the same problems discovered with a token-based method. Specifically, if a task implements a long or infinite loop in its critical section, interrupts may be disabled for an extended period.

The problem can be solved with a variation on the solution developed for the token based scheme. Rather than disabling all interrupts, when the *entry* code segment is entered, all interrupts below a specified level are disabled or masked. A timer that can interrupt at a level above that set by the mask is enabled. If the timer expires, the system can preempt the offending task and handle it as is appropriate for the design.

entry

Once again, an interrupt-based approach meets the requirements for solving the critical section problem. The one caveat is that such an approach will not be effective in a multiprocessor approach utilizing shared memory since we only have the ability to manage interrupts on our own processor.

7.5 Semaphores

A protocol to protect a critical section was suggested by Professor Edsger Wybe Dijkstra, a distinguished computer science pioneer from Rotterdam, The Netherlands. Dr. Dijkstra has made significant contributions to almost every aspect of the field of computing science.

As his solution to the critical section problem, he devised what is called a *semaphore*. In its simplest form, a semaphore is a Boolean variable or an integer– S that can be accessed only through two *atomic* operations:

semaphore

atomic

```
wait - P(S)
signal - V(S)
```

The letters P and V are the first letters of the Dutch words *proberen*, which means *to test*, and *verhogen*, which means *to increment*. At this point in the discussion, the value of a semaphore will reflect whether or not access to the critical variable is available. The word "atomic" qualifying the access operations for the semaphore is important, as was discussed earlier for the await operation.

proberen

to test, verhogen, to increment

The *wait* operation tests the value of the semaphore, and if it is false, sets it to true. The *signal* operation sets the value to false. The wait operation performs its job in two steps: *test*, then *set*. These steps must be seen from outside of the wait as a single, atomic operation.

wait

signal, test

set

The sequence of events diagrammed in Figure 12.34 should not be possible. In the situation presented, the two tasks, T0 and T1, are executing. T0 currently has the CPU and needs to enter the critical section. It executes the *wait*. If the *test* and *set* operation is not atomic, T0 could complete the test portion and see that the resource is available. In the meantime, task T1, which has a higher priority, interrupts and also needs the resource. It,

wait, test, set

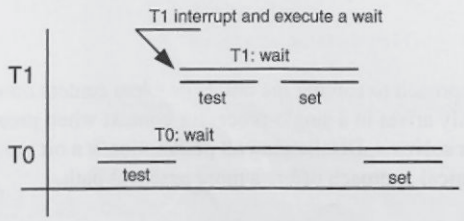


Figure 12.34 A Nonatomic Model of a Flag Used to Protect a Critical Section

wait too, executes the *wait,* which it is allowed to complete. Task T1 then exits, and T0 resumes where it left off and sets the flag. Both processes now believe that they have mutually exclusive access to the critical area.

As long as neither task changes the value of the critical variable, everything will work as expected. However, a write operation by either task can potentially create a serious problem.

The operations may be defined by the code fragments presented in Figure 12.35. Observe the similarity with the await operation.

```
wait(s)                    signal(s)
{                          {
    while (s);                 s = FALSE;
    s = TRUE;              }
}
s initialized to FALSE
```

Figure 12.35 A Model of Semaphore Behavior

await Bear in mind that, as with the *await* control statement, although shown as several steps, the wait must execute as a single, atomic operation. Lest the reader think that the semicolon following while is in error, it is not. Such a construct forces a task to block as long as the semaphore is set.

test, set The *test* and *set* operation (abbreviated in various texts as *TS*, *TAS*, or *TNS*) is implemented as a hardware instruction on many processors.

The semaphore can now be used to protect a critical resource as demonstrated in the two code fragments presented in Figure 12.36.

```
Task T0                    Task T1
{                          {
    ...                        ...
    wait(s)                    wait(s)
    critical section           critical section
    signal(s)                  signal(s)
    ...                        ...
}                          }
```

Figure 12.36 Protecting a Critical Section with a Semaphore

The task that executes the wait(s) first will gain access to the critical section. The second task will block, waiting for the other task to execute the signal. Thereafter, it, too, can proceed.

Process Synchronization

One can use the semaphore in a slightly different way to force the execution order of several asynchronous tasks. For the basic case, consider an application with two such tasks, T0 and T1, which are cooperating on a portion of the application. Task T0 contains a function $f(s_0)$,

```
Task T0                          Task T1
{                                {
    ...                              ...
    f(s1);                           wait(sync);        // wait
    signal(sync);      // signal    g(s2);
    ...                              ...
}                                }
```

Figure 12.37 Using a Semaphore to Control the Order of Execution

and task T1 contains a function, $g(s_1)$. Their execution order is critical; the function $f(s_0)$ must be executed before $g(s_1)$. To achieve such a synchronization, we define the semaphore *sync* and initialize it to TRUE. The code fragments in Figure 12.37 illustrate the design.

Observe that because sync is initialized to TRUE, T1 will execute $g(s_2)$ only after T0 executes statement $f(s_1)$.

2.7.7 Spin Lock and Busy Waiting

wait

busy waiting

spin lock

The one disadvantage of using semaphores for synchronization as we have described earlier is that when a *wait* for a shared resource or event, for example, is encountered, the encountering process is blocked and must loop continuously while waiting. Such a phenomenon is called *busy waiting*. Under such a condition, the waiting processes waste CPU cycles that other processes could use productively. The lock on the critical section is called a *spin lock* because the process spins while waiting for the lock to open. Of course, the advantage of such a lock is that there is no context switch which can take significant time. If the lock is expected to be held for only a short time, the spin lock can be particularly useful in time-critical situations.

2.7.8 Counting Semaphores

binary semaphores

counting semaphores

The semaphores we have looked at are called *binary semaphores*; they can take on either one of two values. The definition can be expanded slightly to permit the semaphore to take on a range of values from 0 to N-1; such semaphores are called *counting semaphores*.

wait

block

waiting

signal

ready

Each such semaphore has an integer value and (potentially) a list of associated processes. When a process executes a *wait* operation and the semaphore is not available, rather than wait the process can *block* itself. Through the block operation, the process places itself in a waiting queue associated with the semaphore. The state of the process is changed to *waiting*, and control is transferred to the scheduler. The blocked process can be restarted when some other task executes a *signal* operation. The restart operation is initiated by a *wakeup* operation that places the task in the *ready* state and into the ready queue. Counting semaphores can be particularly useful when we must manage a pool of identical resources.

The definition of the semaphore operations is modified slightly, as seen in the code fragments in Figure 12.38. Nonetheless, the modeled operation of the semaphore remains atomic. The semaphore now defined as s is initialized to 0.

block, wakeup

Note that the *block* operation suspends the invoking process and the *wakeup* resumes execution of the blocked process. Both operations are provided by operating system calls. Observe that the waiting list can be implemented by a linked list and perhaps implement as FIFO or a priority queue.

```
wait(s)
{
    s = s+1;
    if (s > 1)
    {
        add process to waiting queue;
        block;
    }
}
```

```
signal(s)
{
    s = s-1;
    if (s >1)
    {
        remove process from waiting queue;
        wakeup(p);
    }
}
```

Figure 12.38 A Code Fragment Modeling a Counting Semaphore

12.8 TALKING AND SHARING IN SPACE

So far, we have discussed the problems of sharing, cooperation, and synchronization among asynchronous tasks. Let's look at an application in which we can begin to use these concepts.

12.8.1 The Bounded Buffer Problem

First let's describe the objective. One of the major goals in designing embedded applications is to ensure that they perform in a highly robust manner that tolerates faults and misuse. Consider the following problem.

EXAMPLE 12.1

The application is to build the data management portion of an extensible digital imaging system to be used on the next generation Rovers that will engage in an ongoing exploration of Mars.

The goal of the mission is to conduct a series of detailed studies of the Martian surface and surrounding environment. The system is configured with several cameras that can continuously collect a variety of image data. The data may include infrared scans, atmospheric analysis, or topographic mapping.

The imaging system is mounted on the Rover. Data is collected in a buffer and then uploaded to an orbiting satellite that will subsequently transmit the image data to any one of a number of tracking stations on the Earth.

Because the objective is to map or sample as much of the environment as possible during each mission as data is collected, it is stored into any one of a set of N smaller buffers rather than one large one. With such a scheme, there is no waiting for the one buffer to be emptied before scanning can begin again, thereby maximizing the transfer on both sides. Thus, as each buffer is filled, image data is directed to the next free buffer. So as not to miss communication with one of the various Earth stations, the mother ship must upload the collected data as soon as it becomes available.

The block diagram in Figure 12.39 illustrates the system.

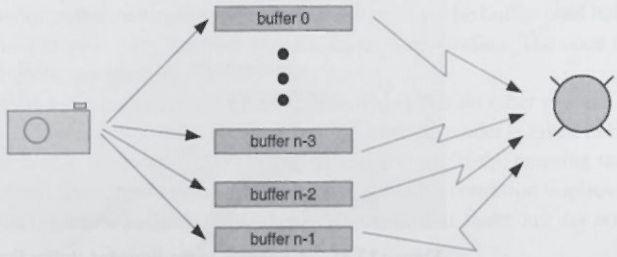


Figure 12.39 Information Sharing Utilizing an n Buffer Design

META-1026, p. 35
Meta v. Resonant
IPR2025-00823

To solve the problem we first identify the essential requirements.

There are a couple of things that must be managed: the count of the number of free/full buffers and controlled access to a specific buffer for reading and writing the image data.

Next, we work on a solution.

The imaging system is a producer of data and the satellite is a consumer. We will use semaphores to manage access to the variables specifying the number of full or empty buffers and thence access to those buffers. To begin, we define the semaphores:

mutex Provides mutual exclusion for accesses to buffer pool—initialized to the value 1

empty Count number of empty buffers—initialized to n-1

full Count number of full buffers—initialized to 0

The algorithm works as follows.

The producer will check to see if there are empty buffers, if so, wait for exclusive access to the buffer pool. Once access is gained, the producer will add the data, then exit. On the consumer side, the consumer will see if any buffers have data available; if so, will wait for exclusive access. When the buffer pool is open, the consumer will retrieve the data and exit.

The producer code fragment is illustrated in Figure 12.40.

```
Task T0–Produce–Rover Side
    while(1)
        ...
        produce an item T0Item
        ...
        wait(empty);                    // wait for available buffer
            wait(mutex);                // buffer available
                                        // wait for exclusive access to buffer pool
            ...
                add T0Item to buffer;   // copy image data to buffer
            ...
            signal(mutex);              // signal buffer pool available
        signal(full);                   // signal data available
        ...
    end while
```

Figure 12.40 A Solution to the Bounded Buffer Problem: The Producer Side

The consumer code fragment is illustrated as shown in Figure 12.41.

```
Task T1–Consume–Satellite Side
    while(1)
        wait(full);                     // wait for data to become available
            wait(mutex);                // wait for exclusive access to buffer pool
            ...
                remove T1Item from buffer;  // retrieve image data
            ...
            signal(mutex);              // signal buffer pool available
        signal(empty);                  // signal date read
        ...
        consume item T1Item             // use image data
        ...
    end while
```

Figure 12.41 A Solution to the Bounded Buffer Problem: The Consumer Side

META-1026, p. 36
Meta v. Resonant
IPR2025-00823

unded Buffer problem

The problem just described is a classic synchronization problem known as the *Bounded Buffer Problem*.

2 The Readers and Writers Problem

A new engineer proposes that since there are a number of buffers, the imaging system can be enhanced by permitting data to be collected from several cameras at the same time and stored in one of the buffers. Also, data can be uploaded using several links and thereby speed up that process as well.

To demonstrate its operation, the engineer quickly puts together a simple model of the system. It works well most of the time, but occasionally data gets corrupted and he or she cannot understand why.

The proposed design exhibits one of the classic problems. We have a data object that must be shared among several concurrent processes. Some may want to upload (read) and others may want to store (write). The processes are referred to as *readers* and *writers*.

readers, writers

When operating, if multiple readers access the data simultaneously, there is no problem. If a writer and any other process access the shared data simultaneously, then there is the potential for a big problem. This problem is referred to as the *readers-writers* problem. There are several variations to the problem.

readers-writers

> *First Readers-Writers*: No reader waits unless a writer has obtained access of shared variable.
>
> *Second Readers-Writers*: Once a writer is ready, it performs the write as soon as possible. If a writer is waiting, no new reader started.

Let's see how the young engineer's problem can be solved. We will present a solution to the first readers-writers problem. To start, we define the following terms.

Semaphores
mutex, wrtSem, both initialized to 1

mutex
Used to ensure mutual exclusion when numReaders is updated

wrtSem
Used to ensure mutual exclusion for writer access

numReaders
Integer count of the number of readers currently accessing the shared buffer pool, initialize to 0

Each writer process must check for exclusive access to the buffer pool before writing. We ensure this by protecting the pool with the semaphore *wrtSem*. The code fragment for the writer is given in Figure 12.42.

wrtSem

As many readers as desired are permitted, provided that no other process is accessing the buffer pool to change the data. The code fragment for the reader is given in Figure 12.43.

Observe that in the entry section of the critical section, if the entering task is not the only reader, then, there must already be other readers. Such a condition implies there cannot be any writers. Otherwise, one must check to ensure that there are no writers before proceeding.

META-1026, p. 37
Meta v. Resonant
IPR2025-00823

```
Writer Process
    wait(wrtSem);                    // wait for wrtSem == 1
                                     // wrtSem = 0
        ...
    // critical section

        perform writing;
        ...
    signal(wrtSem);                  // wrtSem = 1
    ...
```

Figure 12.42 The First Readers and Writers Problem: The Writer Side

```
Reader Process
    while(1)
        wait(mutex);                 // wait while mutex == 1
                                     // mutex = 0
            numReaders++;            // inc number of readers
            if (numReaders ==1)      // if i'm the only reader
                wait(wrtSem);        // make sure no writers
                                     // wrtSem = 1
            end if
        signal(mutex);               // mutex = 0

    // critical section
            ...
        Perform reading;
            ...

        wait(mutex);                 // wait for mutex == 1
                                     // mutex = 0
            numReaders--;            // dec number of readers

            if (numReaders ==0)      // no readers
                signal(wrtSem);      // wrtSem = 0
            end if
        signal(mutex);               // mutex = 0
        ...
    end while
```

Figure 12.43 The First Readers and Writers Problem: The Reader Side

wrtSem
mutex, signal(wrtSem)

If a writer is in the critical section, n readers are waiting, one reader is queued on *wrtSem*, n-1 readers are queued on *mutex*, and if a writer executes *signal(wrtSem)*, it may resume the waiting readers or one waiting writer. The decision is made by the scheduler.

The tacit assumption being made is that the buffers that are being written to or read from are managed to ensure that neither underflow nor overflow occurs.

12.9 MONITORS

The semaphores we have studied are a fundamental method for synchronism. However, they are a low-level mechanism, and it is easy to make errors with them. An alternate solution uses a data type called a *monitor*. Monitors are program modules that offer more structure than semaphores, with an implementation that can be as efficient.

monitor

A monitor is a data abstraction mechanism that encapsulates a representation of an abstract object. The monitor provides a public interface as the only means by which internal data may be manipulated. Note that this is similar to a class in either C++ or Java. The monitor contains an internal (private) variable to store the object's state and procedures (methods or function members) that implement the operations on the object. Mutual exclusion is satisfied by ensuring that procedures in the same monitor cannot execute simultaneously. Conditional synchronization is provided through *condition variables*.

condition variables

A monitor is used to group a representation and implementation of a shared resource. It has an *interface* and a *body*. The (public) interface specifies those operations provided by the resource, while the body contains variables that represent the state of the resource. Internal procedures implement the operations specified in the interface. The monitor can be schematically illustrated as shown in Figure 12.44.

interface, body

```
monitor monName
{
    initialization statements
    procedures
    permanent variables
}
```

Figure 12.44 The Monitor—A Typical Structure

The procedures implement the visible operations. All processes in the monitor share the permanent variables. They are denoted permanent because they retain their values on exit as long as the monitor exists. Such behavior occurs in C or C++ with static variables. The procedures may also have local variables.

By virtue of being an abstract data type (ADT), the monitor is a distinct scope. Only the procedure names are visible outside of the monitor—the public interface. Permanent variables can only be changed through one of the visible procedures. Statements within the monitor cannot affect variables outside the monitor, that is, those in a different scope. Permanent variables are initialized before any procedure is called. The initialization is accomplished by executing initialization procedures when the monitor instance is created.

The major difference between the monitor and a class in C++ or Java is that the monitor is shared by multiple concurrently executing processes or threads. Consequently, the threads or processes using a monitor may require *mutual exclusion* to the monitor variables as well as *synchronization* to ensure that the monitor state is conducive to continued execution.

mutual exclusion
synchronization

Mutual exclusion is usually implicit; synchronization is implemented explicitly. Different processes require different forms of synchronization. The implementation of the necessary synchronization is accomplished through *condition variables*. An external task or thread calls a monitor procedure. The procedure is *active* if a thread or task is executing a statement in the procedure. At most one instance of a monitor procedure is *active* at any one time. The simultaneous invocation of two different procedures or two invocations of the same procedure is not permitted.

condition variables
active

META-1026, p. 39
Meta v. Resonant
IPR2025-00823

By definition, the procedures execute with mutual exclusion that is ensured by the language library and operating system. Mutual exclusion is generally implemented by using locks or semaphores and by inhibiting certain interrupts.

.9.1 Condition Variables

Condition variables are used as part of the synchronization process and are intended to delay a task or thread that cannot safely continue until the monitor's state satisfies some Boolean condition. Note that condition variables are similar to the guard conditions in UML state charts. They are then used to awaken the delayed process once the condition becomes true.

cond

A condition variable is an instance of a variable of type *cond*.

```
cond myCondVar;
```

The declaration can only occur inside the monitor. The value of the condition variable is a queue of delayed processes. Initially, the queue is empty. The value on the queue can only be accessed indirectly, for example, to test its state.

```
empty(myCondVar);
```

A thread can block on a condition variable:

```
wait(myCondVar);
```

wait

Execution of the *wait* causes the task to move to the rear of the queue and to relinquish exclusive access to the monitor. A blocked process is awakened using

```
signal(myCondVar);
```

signal

Execution of a *signal* causes the task at the head of the queue to awaken.

Observe that the execution of *signal* seems to cause a dilemma. Upon execution, two tasks have the potential to execute: the awakened task and the signaling task. Such a situation seems to contradict the requirement that only a single task or thread can be active in the monitor at any one time.

There are two possible paths for resolution:

Signal and Continue nonpreemptive

- *Signal and Continue*—the signaling task continues, and the awakened task resumes at some later time. Such a scheme is considered *nonpreemptive;* the process executing the signal retains exclusive control of the monitor.

Signal and Wait, preemptive

- *Signal and Wait*—is considered to be *preemptive*. The task executing the signal relinquishes control and passes the lock to the awakened task. The awakened process preempts the signaling process.

The process is described in Figure 12.45.

calls
entry queue
return, wait

The operation/synchronization occurs as follows. A task *calls* a monitor procedure. If another task is executing in the monitor, the caller is placed into the *entry queue*. When the monitor becomes free, as a result of a *return* or *wait,* one task moves from the entry queue into the monitor.

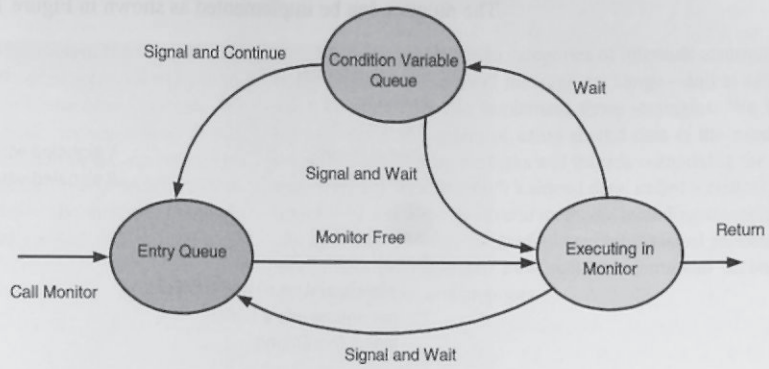


Figure 12.45 A State Diagram Model for a Monitor

If no other tasks are executing, the calling task passes through the entry queue and begins executing immediately. If the task executes *wait* on a condition variable while executing in the monitor, it enters the queue associated with that variable.

Signal and Continue entry, Signal and Wait

When the task executes a *Signal and Continue* on a condition variable, the task at the head of the associated queue now moves to the *entry* queue. If a task executes a *Signal and Wait* on a condition variable, the task at the head of the associated queue moves to the monitor and the task executing in the monitor moves to the entry queue.

12.9.2 Bounded Buffer Problem with Monitor

Let's revisit the bounded buffer problem and implement the design with a monitor. As before, there is a pool of n buffers. We will assume that each can hold one item.

Define a monitor *boundedBuffer*.

Define the following condition variables:

notEmpty

Signaled when buffer count > 0

Tracks empty buffers, initialized to 0

notFull

Signaled when buffer count < n-1

Tracks full buffers, Initialized to 0

Define the procedures:

put(data)

Puts data into a buffer when space available

get(data)

Gets data from a buffer when data available

Define the protected entity:

bufferPool

The monitor can be implemented as shown in Figure 12.46.

```
monitor boundBuffer
    bufferPool;
    count = 0;
    cond notEmpty;                    // signaled when count > 0
    cond notFull;                     // signaled when count < n

    put(anItem)
    {
        while(count == n) wait (notFull);
        put anItem into a buffer
        signal (notEmpty);
    }

    get(anItem)
    {
        while(count == 0) wait (notEmpty);
        get anItem from a buffer
        signal (notFull);
    }
```

Figure 12.46 A Monitor Solution to the Bounded Buffer Problem

Code fragments for the implementation are illustrated in Figure 12.47.

```
Producer                          | Consumer
    while(1)                      |     while(1)
        ...                       |         ...
        produce item anItem       |         boundBuffer.get(anItem)
        ...                       |         ...
        boundBuffer.put(anItem)   |         consume item anItem
        ...                       |         ...
    end while                     |     end while
```

Figure 12.47 Using the Monitor in the Producer and the Consumer to Solve the Bounded Buffer Problem

12.10 STARVATION

When working with semaphores and monitors, a potential problem called *starvation* exists. That is, one process is permanently prevented from running. Such a situation can occur when a process is waiting within a monitor or semaphore and other processes are added or removed in LIFO order.

12.11 DEADLOCKS

deadlock

When working in a multitasking environment, one can create a second problem called a *deadlock*. A deadlock occurs when each process in a set of processes needs resources that are held by other processes in that set in order to continue. We will study the deadlock problem and examine several possible solutions in depth in the next chapter.

12.12 SUMMARY

In this chapter we continued the discussion of time and the critical role it plays in the design of embedded applications by introducing the concepts of *reactive* and *time-based systems*. We have studied, in some detail, the basic responsibilities of task scheduling and intertask communication in the operating system. We have examined a number of different criteria for assessing scheduling algorithms; we learned the difference between static and dynamic scheduling, and we looked at several algorithms in each category.

We have looked at two categories of intertask communication—shared variables and message exchange—and at several ways by which we can implement those strategies. We have learned that a side effect of using shared data is the need for coordinated access by the tasks and threads comprising the system. We have seen that such a shared data, called a critical section, can be managed by several methods, including semaphores and monitors. Finally, we studied several classical models for shared data problems and how such problems can be solved using semaphores and monitors.

12.13 REVIEW QUESTIONS

Time, Time-Based Systems, Reactive Systems

12.1 What is the difference between an interval and a duration?

12.2 What is a time-based embedded system? a reactive embedded system?

12.3 What is the difference between a periodic and an aperiodic event or operation?

12.4 Explain what is meant by delay in an embedded application; by jitter.

12.5 What is meant by the expressions *hard* or *hard deadline* in a real-time embedded context?

12.6 What is firm real-time? soft real-time?

Scheduling

12.7 What is meant when a task is said to be schedulable? deterministically schedulable?

12.8 What is CPU utilization? Why is it important?

12.9 When are scheduling decisions made?

12.10 What is the difference between a preemptive and a nonpreemptive system?

12.11 Several scheduling criteria were outlined in the chapter. What are these?

12.12 What are the different scheduling algorithms identified in the chapter?

12.13 What is deterministic modeling? a queuing model?

12.14 What is simulation? emulation? What is the difference between them?

Intertask Communication

12.15 What are the three primary components that make up the intertask communication model introduced in this chapter?

12.16 One method introduced in the chapter for exchanging information between tasks was called shared variables. What does this mean?

12.17 Message exchange was introduced as another means by which information might be exchanged between tasks in an embedded application. What does this mean?

12.18 What is a rendezvous in a message exchange model?

12.19 What is a buffer in a message exchange model?

Task Cooperation, Synchronization, and Sharing

12.20 What is a critical section?

12.21 Describe what is meant by the entry and exit sections with respect to a critical section.

12.22 What requirements must be met in order to solve a critical section problem?

12.23 What is meant by the expression *atomic operation*?

12.24 What does the expression *test and set* mean?

12.25 What is a semaphore?

12.26 Discuss how a semaphore can be used to solve the critical section problem.

12.27 What is a spin lock?

12.28 What is a counting semaphore?

12.29 What is the bounded buffer problem?

12.30 What is the readers and writers problem?

12.31 What is a monitor?

12.32 How does a monitor meet the specified requirements for solving a critical section problem?

12.33 What is starvation?

12.34 What is a deadlock?

12.14 THOUGHT QUESTIONS

Time, Time-Based Systems, Reactive Systems

12.1 What is the difference between absolute time and relative time? Give two examples of each in an embedded application.

12.2 Give two examples of periodic and aperiodic events or operations in an embedded application.

Scheduling

12.3 Give an example of an embedded application for which each of the scheduling criteria discussed in the chapter might be best suited. Explain and justify your answer.

12.4 The chapter introduces several different scheduling algorithms. For each algorithm presented, give an example of an embedded application for which the algorithm might be best suited. Explain and justify your answer.

Intertask Communication

12.5 The chapter introduced several shared variable models. Identify each of these and explain how each works.

12.6 For each of the shared variable models, identify a strength and a weakness.

12.7 Give an example of an embedded application in which each of the shared variable models might be used. Explain and justify your choice.

12.8 Explain how message exchange as a means for exchanging information between tasks in an embedded application might work.

12.9 Discuss the advantages and disadvantages of message exchange versus shared variables in an embedded application.

12.10 Explain the difference between direct and indirect communication in a message exchange model? Give an example of each and explain the pros and cons of each approach in your selected applications.

12.11 Explain the difference between symmetric and asymmetric addressing in a message exchange model. Give an example of each and explain the pros and cons of each approach in your selected applications.

12.12 Several different buffering schemes were introduced. What were these? Give several advantages and disadvantages of each approach.

12.13 Give an example of an embedded application in which each of the buffering schemes might be used. Explain and justify your choice.

Task Cooperation, Synchronization, and Sharing

12.14 Give an example of a critical section in an embedded application and explain why it exists.

12.15 Why should a *test and set* operation be atomic?

12.16 The chapter presents several alternate solutions to the critical section problem. Describe each and discuss its advantages and disadvantages.

12.17 Discuss the advantages and disadvantages of using a counting versus binary semaphore in embedded applications.

12.18 Give several examples of embedded applications in which a binary or counting semaphore is used. Explain and justify your choice in each case.

12.19 What real-world problem is the bounded buffer problem modeling?

12.20 Give several examples of embedded applications containing a bounded buffer problem.

12.21 What real-world problem is the readers and writers problem modeling?

12.22 Give several examples of embedded applications containing a readers and writers problem.

12.23 How does a monitor differ from a binary semaphore? counting semaphore?

12.24 Explain the purpose of condition variables in a monitor.

12.15 PROBLEMS

12.1 Present a UML sequence diagram to illustrate the behavior of an embedded design comprising four tasks in the polled set.

12.2 Complete the design of the basic polled algorithm given in Figure 12.4 for a system with four tasks in the polled set. Model each task as a mod N_i counter that is incremented each time the task is polled.

12.3 You have a digital event, a positive transition on a signal line, that you must respond to within 40 μ sec. As the designer, you need to determine the best way to handle such a signal. You have two choices, polling or an interrupt. You are in a design review and must present a case justifying one or the other.

(a) Present the pros and cons of polling.

(b) Present the pros and cons of an interrupt-based scheme.

(c) For a polled scheme, give a detailed description of necessary steps prior to polling, during polling, and after the event occurs. Be specific.

(d) For an interrupt-based scheme, give a detailed description of necessary steps prior to the interrupt, during the interrupt, and after the interrupt has been handled. Be specific.

(e) What happens in both cases (polled and interrupt) if all interrupts are globally disabled?

(f) What happens in the interrupt case if no ISR is set up at the interrupt vector location?

12.4 You have a task that must respond to an external event at five different times during a cycle. For two of the times, t_2 and t_3, the response is considered hard real-time and for three of the times, t_0, t_1, t_4, the response is considered soft real-time as shown in Figure P12.48.

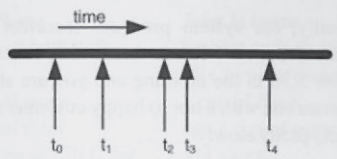


Figure P12.48

As the designer, you can choose only one of the following methods to accommodate the external event: polled, interrupt, or polling an interrupt. Discuss the advantages and disadvantages of each method.

12.5 Design an embedded system to control a traffic light utilizing a state-based schedule. Each direction supports a left turn (right turn if traffic normally drives on the left-hand side) and pedestrian-activated crosswalk control.

12.6 Design an embedded system to control a portable personal entertainment system utilizing a state-based schedule. The system must support the ability to: turn on / select a song to play, play the song, suspend playing, replay a song, turn off.

12.7 Implement a *first-come first-served* scheduling algorithm utilizing a doubly linked list based task queue.

12.8 Repeat Problem 12.7 for a *shortest job first* scheduling algorithm.

12.9 Repeat Problem 12.7 for a *round robin* scheduling algorithm.

12.10 An embedded system has three processes with the following execution times and periods: P1(4, 16), P2(3, 12), P3(2, 8).

(a) What is the CPU utilization for such a system?

(b) Can the set of tasks be scheduled using a rate-monotonic schedule?

(c) If the set of tasks can be scheduled, give the UML sequence diagram for the schedule.

12.11 An embedded system has three processes with the following execution times and periods: P1(4, 16), P2(3, 8), P3(2, 7).

(a) What is the CPU utilization for such a system?

(b) Can the set of tasks be scheduled using a rate-monotonic schedule?

(c) If not, what changes would have to be made to enable the set of tasks to be scheduled using a rate-monotonic schedule?

12.12 An embedded system has five processes with the following execution times and periods: P1(5, 40), P2(5, 60), P3(4, 16), P4(6, 48), P5(12, 96).

(a) What is the CPU utilization for such a system?

(b) Can the set of tasks be scheduled using a rate-monotonic schedule?

(c) If the set of tasks can be scheduled, give the UML sequence diagram for the schedule.

12.13 An embedded system has three processes with the following execution times and periods: P1(4, 16), P2(3, 8), P3(2, 7).

(a) What is the CPU utilization for such a system?

(b) Can the set of tasks be scheduled using an *earliest deadline* schedule?

(c) If the set of tasks can be scheduled, give the UML sequence diagram for the schedule.

12.14 Provide a C algorithm to schedule a set of three tasks using an *earliest deadline* schedule.

12.15 Repeat Problem 12.14 for a *least laxity* schedule.

12.16 An embedded system has the following three jobs, processes, and resources. Devise a schedule using the *shortest job first* algorithm that will achieve optimum utilization of resources and system throughput.

3 Jobs:	J1, J2, J3		
3 Resources:	A/D		
3 Processes:	Measure		M
	CPU	Compute	C
	I/O	Output	O

J1	Time Units	J2	Time Units	J3	Time Units
M1	1	M1	2	M1	3
C1	1	C1	3	C1	3
M2	2	M2	1	M2	2
C2	3	C2	2	C2	2
O1	3	M3	2	M3	3
M3	2	C3	3	C3	3
C3	1	O1	2	O1	2
O2	1				
Total	14		15		18

12.17 Repeat Problem 12.16 using a *rate-monotonic* schedule.

12.18 Repeat Problem 12.16 using an *earliest deadline* schedule.

META-1026, p. 45
Meta v. Resonant
IPR2025-00823

12.19 An embedded application is designed as three tasks. The requirements for each are given in the following table.

Task	Priority	Period	Time Units
1	1	7	2
2	2	16	4
3	3	31	7

(a) Can the three tasks be scheduled using a nonpreemptive scheduling scheme? Why or why not? If so, show the schedule using a UML sequence diagram.

(b) Can the three tasks be scheduled using a preemptive scheduling scheme? Why or why not? If so, show the schedule using a UML sequence diagram.

(c) Can the three tasks be scheduled using a time slice scheduling scheme? Why or why not? If so, what is the value of the time slice to ensure minimum average wait time for all three tasks. Show the schedule using a UML sequence diagram.

12.20 Give a UML class diagram for a buffer that can be shared between two tasks.

12.21 Provide a C implementation of the buffer specified by the class diagram in Problem 12.20.

12.22 Provide a Verilog model of the buffer specified by the class diagram in Problem 12.20.

12.23 Give a UML class diagram for a ping-pong buffer that can be shared between two tasks.

12.24 Give a UML sequence diagram for the operation of a ping-pong buffer.

12.25 Provide a C implementation of the ping-pong buffer specified by the class diagram in Problem 12.24.

12.26 Provide a Verilog model of the ping-pong buffer specified by the class diagram in Problem 12.24.

12.27 Give a UML class diagram for a ring buffer that can be shared between two tasks.

12.28 Provide a C implementation of the ring buffer specified by the class diagram in Problem 12.27.

12.29 Provide a Verilog model of the ring buffer specified by the class diagram in Problem 12.27.

12.30 A shared memory scheme is to be used as a means of exchanging blocks of data between two tasks, T_0 and T_1. The number of blocks of data to be exchanged and their location is not fixed.

(a) Give a data/control flow diagram for the shared memory system.

(b) Explain how your memory system works using a UML sequence diagram and by describing a complete cycle that includes the following: Write by T_0—Read by T_1—Write by T_1—Read by T_0. Be certain to explain how each task knows when and how much to read or write.

(c) How would your design change if three tasks were involved in the exchange?

12.31 As the chief engineer for *Make Me Rich Consultancy*, you have been hired by a start-up embedded systems company *Inside Your Stuff, Ltd.* It seems that they have designed (in less than two weeks) a hard real-time control system for *Fastern Yours Processes, Etc.* The control system supports the following two operations on a collection of data items, $a_0, a_1, a_2 \ldots a_{n-1}$,

get (i)—Returns the value of a_i

put (i, aValue)—Assigns a Value to a_i

The control system has three asynchronous processes that must perform the following transactions:

```
p0: x = read (j); y = read (i); write (j, 52); write (i, 27);
p1: x = read (k); write (i, 43); y = read (j); write (k, 72);
p2: write (k, 25); x = read (i); y = read (j); write (i, 27);
```

Occasionally, the system produces incorrect results and *Fastern Yours Processes, Etc.* is threatening to return the system. It is now 3:30 in the morning and you are at the *Fastern Yours Processes* site with a not so happy customer and a system that is running pretty slowly.

(a) When *Inside Your Stuff, Ltd.* said they had designed a hard real-time system, what did they mean?

(b) Can you identify the problem and explain why it is occurring?

(c) Can you propose a fix? Explain why your solution will solve the problem?

12.32 A colleague has built a simulation of a portion of a telecommunications block. He explains that the system uses a shared buffer that accepts blocks of characters from a measurement process P1 and forward blocks of data to the output process, P2. He has written the following routines, one for P1 and one for P2.

```
full = 0
max = buffer size
p1Generate( )
{
   while (full < max)
   {
      buffer(head) = anItem;
      (head = head + 1) mod max;
      full++;
   }
}

full = 0
max = buffer size
p2Transmit( )
   {
   while (full > 0)
   {
      anItem = buffer(head);
      (head = head - 1) mod max;
      full--;
   }
}
```

META-1026, p. 46
Meta v. Resonant
IPR2025-00823

Occasionally the system either loses data or forwards incorrect data.

(a) Can you explain why?

(b) Please propose (in detail) a way to fix the problem. Modify the existing code as necessary.

(c) Show how your design solves the problem.

12.33 In the pastry corner of the kitchen of a small restaurant, we find two world-class chefs, grumpy Pierre des Oeufs and Jean "la loupe" Farouche, who despise each other. Nonetheless, they must work in the same place and share the same resources. Each is responsible for a different kind of cookie. Here are the recipes:

Grumpy Pierre	*Jean la loupe*
Mix 1 cup of milk with	Preheat oven to 190 C
2 eggs	Mix 1 cup of water with
add 1 cup of sugar	1 cup of flour
add 1 cup of flour	add 1 cup of sugar
Bake in oven at 170 C for	add 1 egg
10 minutes	Bake in oven for 5 minutes

In the kitchen, we have,

One giant carton of milk

One giant crate of eggs

Two large sugar bowls

One large container of flour

One cold water tap

One small oven that has space for one batch of cookies

The previous consultant who tried to schedule the work of Pierre and Jean had a sudden job change to Cinque Terre on the Italian Riviera where he now spends his days sun-drying porcini mushrooms.

Your predecessor was actually quite clever and modeled the two chefs as processes. You find the following bits of code (encrusted with cookie dough) and partially implemented chef processes. Please complete the design.

You have the following nonatomic (they can be interrupted) subroutines available:

```
getEggs( numEggs )          // retrieves numEggs from the crate
                            // of eggs
getFlour( numCups )         // retrieves numCups from the flour
                            // container
getMilk( numCups )          // retrieves and pours numCups from
                            // milk carton
getSugar( numCups )         // retrieves numCups from the
                            // sugar bowl
getWater( numCups )         // retrieves numCups from the tap
putIntoOven( numMinutes )// puts cookie tray into oven for
                            // numMinutes
setOvenTemp( numDegrees )// sets oven temperature to
                            // numDegrees
```

Initialize the following semaphores:

```
Semaphore eggCrate =
Semaphore flourContainer =
Semaphore sugarBowl =
Semaphore waterTap =
Semaphore oven =
Semaphore milkCarton =
```

Complete the two chef processes:

```
process grumpyPierre( )
{
}
process jeanlaLoup( )
{
}
```

12.34 A now defunct engineering firm was hired to design the switching system in a small town railway station. Their final design appears as shown in Figure P12.49.

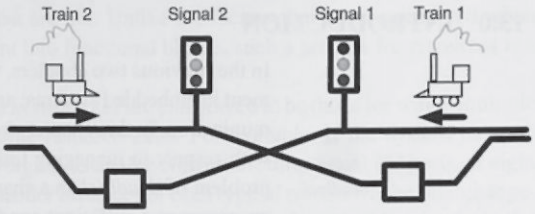


Figure P12.49

Signals 1 and 2 may be Red, Yellow, or Blue.

If Train 1 is approaching Platform 1, it must turn Signal 2, then Signal 1, to *Blue* before proceeding. Similarly, if Train 2 is approaching Station 2, it must turn Signal 1, then Signal 2 to *Red*.

A train may only change the signal (to *Red* or *Blue*) if the signal is in the *Yellow* state.

When Train 1 leaves Station 1, it must turn Signal 1, then Signal 2, to *Yellow*. Similarly, when Train 2 leaves Station 2, it must turn Signal 2, then Signal 1, to *Yellow*.

(a) Are there any problems with the scheme described above? If so, identify what they are.

(b) Will such a scheme prevent collisions? Justify your answer. If not, propose a solution that will.

(c) Will such a scheme prevent deadlocks? Justify your answer. If not, propose a solution that will.

META-1026, p. 47
Meta v. Resonant
IPR2025-00823

Chapter 13

Deadlocks

THINGS TO LOOK FOR . . .

- Scheduling tasks and resource management.
- The problem of deadlock in a shared resource environment.
- The necessary and sufficient conditions for deadlock to occur.
- How to prevent, avoid, and detect deadlocks.
- How to recover from a deadlock state.

13.0 INTRODUCTION

In the previous two chapters, we have addressed several important aspects of task management in embedded systems; among these were scheduling task execution and intertask communication. In this chapter, we will examine aspects of the scheduling and dispatch of tasks with respect to managing task demands for resources. To that end, we will introduce the problem of *deadlock* in a shared-resource, multitasking environment. We will identify the necessary and sufficient conditions for deadlock to occur. First, we examine ways to prevent or avoid deadlock, and then we study methods for detecting a deadlock if, despite best efforts, a deadlock does occur. We conclude by presenting several techniques for recovering from a deadlock state.

deadlock

13.1 SHARING RESOURCES

A multitasking or multiprocessing embedded system has a finite number of resources such as timers, analog-to-digital converters, digital-to-analog converters, and I/O ports. Often several tasks may compete for those resources. When such a request is made and if the requested resources are not available, the task or processor blocks. The implementation of a semaphore or monitor with waiting queue, for example, can result in a situation in which two or more processes wait indefinitely. Such a situation is called a *deadlock*.

Consider the following simple problem in which there are two tasks, T0 and T1, and two resources, R1 and R2. Let each task have two counting semaphores, S0 and S1. Furthermore, let each need both resources to execute its job. Now, let

```
T0 set wait(S0)    // wait for R1 increment S0 (= 1)
T1 set wait(S1)    // wait for R2 increment S1 (= 1)
```

Now let

```
T0 set wait(S1)    // wait for R2 increment S1 (= 2)
T1 set wait(S0)    // wait for R1 increment S0 (= 2)
```


META-1026, p. 48
Meta v. Resonant
IPR2025-00823

The system is now stuck; neither process can continue.

Today the problem of deadlocks is treated rather casually. As systems become more complex and the number of tasks and threads increases, the problem will have to be addressed.

13.2 SYSTEM MODEL

To begin, we formulate a model of the deadlock problem. Any embedded system has a limited number of resources. On one hand, if all systems were architected as a single task or if all the tasks in a multiple-task system have mutually exclusive resource demands, deadlocks cannot occur. On the other hand, for most designs, as tasks enter the system, they are going to need those resources. If the system is going to support preemptive multitasking, those resources will have to be shared. Making this same statement another way, one can say that from the perspective of a single task, a deadlock is not a problem. When analyzing deadlocks—their cause, prevention, detection, and correction—the problem must be considered from a system level. One must take into consideration *all* of the tasks in the system.

tasks
resources

A first high-level model decomposes the problem into two pieces: a set of *tasks* and a set of *resources*. Tasks are largely equivalent; resources are not. One can, therefore, form a coarse-grained partition on the set of resources. One possible partition decomposes the set into two groups—those that are identical and those that are not. Although such a decomposition seems reasonable, one must quantify what constitutes identical resources and what distinguishes them from those that are not. Unlike the factors that were considered when decomposing a problem statement into functional blocks, such a process for resources is a bit more straightforward.

identical resources
dissimilar resources

For the current model, *identical resources* are considered to be those for which multiple interchangeable copies of the same resource exist. For example, if the system has two analog-to-digital converters, two digital-to-analog converters, three serial I/O ports, or eight memory buffers, then one can consider instances of each type of resource to be interchangeable. Allocation of any one to a task may be sufficient. On the other hand, *dissimilar resources* are those that are unique for one reason or another. Of these, for example, there may be only a single copy such as the highest priority interrupt or a single serial I/O port. The current state of the model can be expressed graphically as in Figure 13.0.

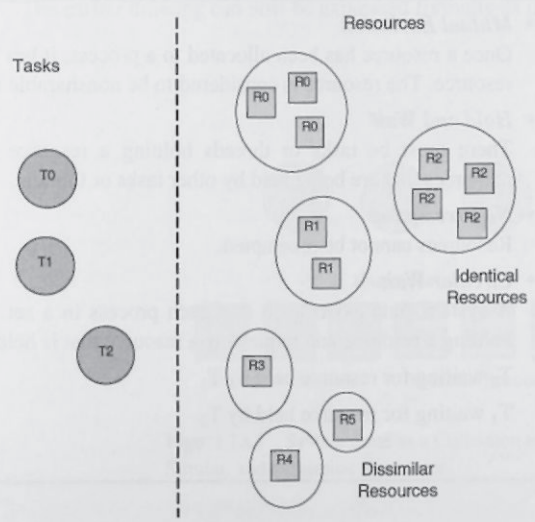


Figure 13.0 System State as a Collection of Tasks and Resources