UNIT-II

1. Briefly description history of os.

In the early days of computing, developers created software applications that included low-level machine code to initialize and interact with the system's hardware directly. This tight integration between the software and hardware resulted in non-portable applications. A small change in the hardware might result in rewriting much of the application itself. Obviously, these systems were difficult and costly to maintain.

As the software industry progressed, operating systems that provided the basic software foundation for computing systems evolved and facilitated the abstraction of the underlying hardware from the application code. In addition, the evolution of operating systems helped shift the design of software applications from large, monolithic applications to more modular, interconnected applications that could run on top of the operating system environment.

Over the years, many versions of operating systems evolved. These ranged from general-purpose operating systems (GPOS), such as UNIX and Microsoft Windows, to smaller and more compact real-time operating systems, such as VxWorks. Each is briefly discussed next.

In the 60s and 70s, when mid-sized and mainframe computing was in its prime, UNIX was developed to facilitate multi-user access to expensive, limited-availability computing systems. UNIX allowed many users performing a variety of tasks to share these large and costly computers. multi-user access was very efficient: one user could print files, for example, while another wrote programs. Eventually, UNIX was ported to all types of machines, from microcomputers to supercomputers.

In the 80s, Microsoft introduced the Windows operating system, which emphasized the personal computing environment. Targeted for residential and business users interacting with PCs through a graphical user interface, the Microsoft Windows operating system helped drive the personal-computing era.

Later in the decade, momentum started building for the next generation of computing: the post-PC,
embedded-computing era. To meet the needs of embedded computing, commercial RTOSes, such as VxWorks, were developed. Although some functional similarities exist between RTOSes and GPOSes, many important differences occur as well. These differences help explain why RTOSes are better suited for real-time embedded systems.
Some core functional similarities between a typical RTOS and GPOS include:

- some level of multitasking,
- software and hardware resource management,
-

provision of underlying OS services to applications, and

- abstracting the hardware from the software application.

On the other hand, some key functional differences that set RTOSes apart from GPOSes include:

- better reliability in embedded application contexts the ability to scale up or down to meet application needs,
- faster performance,
- reduced memory requirements,
- scheduling policies tailored for real-time embedded systems,
- support for diskless embedded systems by allowing executables to boot and run from ROM or RAM, and
- better portability to different hardware platforms.

Today, GPOSes target general-purpose computing and run predominantly on systems such as personal computers, workstations, and mainframes. In some cases, GPOSes run on embedded devices that have ample memory and very soft real-time requirements. GPOSes typically require a lot more memory, however, and are not well suited to real-time embedded devices with limited memory and high performance requirements.

RTOSes, on the other hand, can meet these requirements. They are reliable, compact, and scalable, and they perform well in real-time embedded systems. In addition, RTOSes can be easily tailored to use only those components required for a particular application.

Again, remember that today many smaller embedded devices are still built without an RTOS. These simple devices typically contain a small-to-moderate amount of application code. The focus of this book, however, remains on embedded devices that use an RTOS.

2. Construct neat diagram, typical structure of RTO’S.

In the simplest form, real-time systems can be defined as those systems that respond to external events in a timely fashion, as shown in Figure 1.5. The response time is guaranteed. We revisit this definition after presenting some examples of real-time systems.

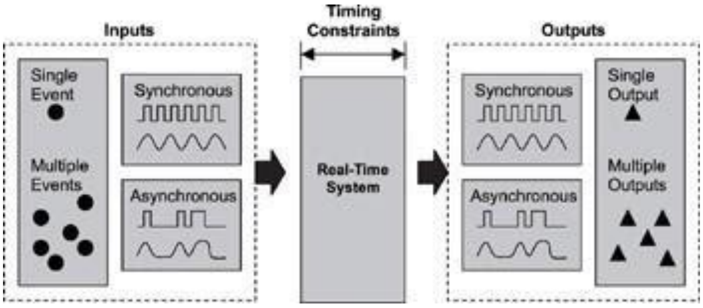


Figure 1.5: A simple view of real-time systems.

External events can have synchronous or asynchronous characteristics. Responding to external events includes recognizing when an event occurs, performing the required processing as a result of the event, and outputting the necessary results within a given time constraint. Timing constraints include finish time, or both start time and finish time.

A good way to understand the relationship between real-time systems and embedded systems is to view them as two intersecting circles, as shown in Figure 1.6. It can be seen that not all embedded systems exhibit real-time behaviors nor are all real-time systems embedded. However, the two systems are not mutually exclusive, and the area in which they overlap creates the combination of systems known as *real-time embedded systems*.

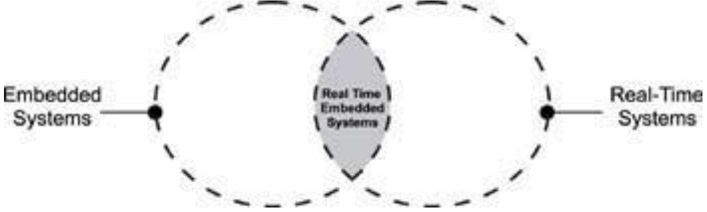
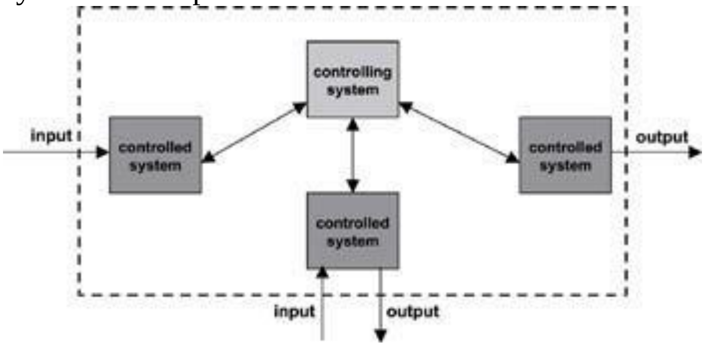


Figure 1.6: Real-time embedded systems.

Knowing this fact and because we have covered the various aspects of embedded systems in the previous sections we can now focus our attention on real-time systems



1. Real-Time Systems

3. Explain the characteristics of RTO'S.

The C&D system in the weapons defense system must calculate the anticipated flight path of the incoming missile quickly and guide the firing system to shoot the missile down before it reaches the destroyer. Assume T1 is the time the missile takes to reach the ship and is a function of the missile's distance and velocity. Assume T2 is the time the C&D system takes to activate the weapons firing control system and includes transmitting the firing coordinates plus the firing delay. The difference between T1 and T2 is how long the computation may take. The missile would reach its intended target if the C&D system took too long in computing the flight path. The missile would still reach its target if the computation produced by the C&D system was inaccurate. The navigation system in the cruise missile must respond to the changing terrain fast enough so that it can re-compute coordinates and guide the altitude control system to a new flight path. The missile might collide with a mountain if the navigation system cannot compute new flight coordinates fast enough, or if the new coordinates do not steer the missile out of the collision course.

Therefore, we can extract two essential characteristics of real-time systems from the examples given earlier. These characteristics are that real-time systems must produce correct computational results, called *logical or functional correctness*, and that these computations must conclude within a predefined period, called *timing correctness.*

Real-time systems are defined as those systems in which the overall correctness of the system depends on both the functional correctness and the timing correctness. The timing cor-rectness is at least as important as the functional correctness.

It is important to note that we said the timing correctness is at least as important as the functional correctness. In some real-time systems, functional correctness is sometimes sacrificed for timing correctness. We address this point shortly after we introduce the classifications of real-time systems.

Similar to embedded systems, real-time systems also have substantial knowledge of the environment of the controlled system and the applications running on it. This reason is one why many real-time systems are said to be deterministic, because in those real-time systems, the response time to a detected event is bounded. The action (or actions) taken in response to an event is known a priori. A deterministic real-time system implies that each component of the system must have a deterministic behavior that contributes to the overall determinism of the system. As can be seen, a deterministic real-time system can be less adaptable to the changing environment. The lack of adaptability can result in a less robust system. The levels of determinism and of robustness must be balanced. The method of balancing between the two is system- and application-specific. This discussion, however, is beyond the scope of this book.
Consult the reference material for additional coverage on this topic.

2. Hard and Soft Real-Time Systems

In the previous section, we said computation must complete before reaching a given deadline. In other words,
real-time systems have timing constraints and are deadline-driven. Real-time systems can be classified, therefore, as either hard real-time systems or soft real-time systems.

What differentiates hard real-time systems and soft real-time systems are the degree of tolerance of missed deadlines, usefulness of computed results after missed deadlines, and severity of the penalty incurred for failing to meet deadlines.

For hard real-time systems, the level of tolerance for a missed deadline is extremely small or zero tolerance. The computed results after the missed deadline are likely useless for many of these systems. The penalty incurred for a missed deadline is catastrophe. For soft real-time systems, however, the level of tolerance is non-zero. The computed results after the missed deadline have a rate of depreciation. The usefulness of the results does not reach zero immediately passing the deadline, as in the case of many hard real-time systems. The physical impact of a missed deadline is non-catastrophic.

A *hard real-time system* is a real-time system that must meet its deadlines with a near-zero degree of flexibility. The deadlines must be met, or catastrophes occur. The cost of such catastrophe is extremely high and can involve human lives. The computation results obtained after the deadline have either a zero-level of usefulness or have a high rate of depreciation as time moves further from the missed deadline before the system produces a response.

A *soft real-time system* is a real-time system that must meet its deadlines but with a degree of flexibility. The deadlines can contain varying levels of tolerance, average timing deadlines, and even statistical distribution of response times with different degrees of acceptability. In a soft real-time system, a missed deadline does not result in system failure, but costs can rise in proportion to the delay, depending on the application.

Penalty is an important aspect of hard real-time systems for several reasons.

- What is meant by 'must meet the deadline'?

- It means something catastrophic occurs if the deadline is not met. It is the penalty that sets the requirement.

- Missing the deadline means a system failure, and no recovery is possible other than a reset, so the deadline must be met. Is this a hard real-time system?

  That depends. If a system failure means the system must be reset but no cost is associated with the failure, the deadline is not a hard deadline, and the system is not a hard real-time system. On the other hand, if a cost is associated, either in human lives or financial penalty such as a $50 million lawsuit, the deadline is a hard deadline, and it is a hard real-time system. It is the penalty that makes this determination.

- What defines the deadline for a hard real-time system?

- It is the penalty. For a hard real-time system, the deadline is a deterministic value, and, for a soft real-time system, the value can be estimation.

One thing worth noting is that the length of the deadline does not make a real-time system hard or soft, but it is the requirement for meeting it within that time.

The weapons defense and the missile guidance systems are hard real-time systems. Using the missile guidance system for an example, if the navigation system cannot compute the new coordinates in response to approaching mountain terrain before or at the deadline, not enough distance is left for the missile to change altitude. This system has zero tolerance for a missed deadline. The new coordinates obtained after the deadline are no longer useful because at subsonic speed the distance is too short for the altitude control system to navigate the missile into the new flight path in time. The penalty is a catastrophic event in which the missile collides with the mountain. Similarly, the weapons defense system is also a zero-tolerance system. The missed deadline results in the missile sinking the destroyer, and human lives potentially being lost. Again, the penalty incurred is catastrophic.

On the other hand, the DVD player is a soft real-time system. The DVD player decodes the video and the audio streams while responding to user commands in real time. The user might send a series of commands to the DVD player rapidly causing the decoder to miss its deadline or deadlines. The result or penalty is momentary but visible video distortion or audible audio distortion. The DVD player has a high level of tolerance because it continues to function. The decoded data obtained after the deadline is still useful.

Timing correctness is critical to most hard real-time systems. Therefore, hard real-time systems make every effort possible in predicting if a pending deadline might be missed. Returning to the weapons defense system, let us discuss how a hard real-time system takes corrective actions when it anticipates a deadline might be missed. In the weapons defense system example, the C&D system calculates a firing box around the projected missile flight path. The missile must be destroyed a certain distance away from the ship or the shrapnel can still cause damage. If the C&D system anticipates a missed deadline (for example, if by the time the precise firing coordinates are computed, the missile would have flown past the safe zone), the C&D system must take corrective action immediately. The C&D system enlarges the firing box and computes imprecise firing coordinates by methods of estimation instead of computing for precise values. The C&D system then activates additional weapons firing systems to compensate for this imprecision. The result is that additional guns are brought online to cover the larger firing box. The idea is that it is better to waste bullets than sink a destroyer.

This example shows why sometimes functional correctness might be sacrificed for timing correctness for many real-time systems.

Because one or a few missed deadlines do not have a detrimental impact on the operations of soft real-time systems, a soft real-time system might not need to predict if a pending deadline might be missed. Instead, the soft real-time system can begin a recovery process after a missed deadline is detected.

For example, using the real-time DVD player, after a missed deadline is detected, the decoders in the DVD player use the computed results obtained after the deadline and use the data to make a decision on what future video frames and audio data must be discarded to re-synchronize the two streams. In other words, the decoders find ways to catch up.

So far, we have focused on meeting the deadline or the finish time of some work or job, e.g., a computation. At times, meeting the start time of the job is just as important. The lack of required resources for the job, such as CPU or memory, can prevent a job from starting and can lead to missing the job completion deadline. Ultimately this problem becomes a resource-scheduling problem. The scheduling algorithms of a real-time system must schedule

system resources so that jobs created in response to both periodic and aperiodic events can obtain the resources at the appropriate time. This process affords each job the ability to meet i meet iic timing constraints. This topic is addressed

4.Explain the issues of synchronization communication and concurrency.

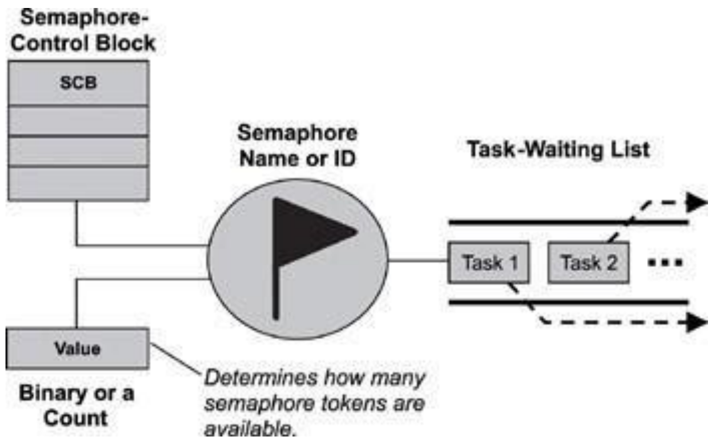
Tasks synchronize and communicate amongst themselves by using *intertask primitives* , which are kernel objects that facilitate synchronization and communication between two or more threads of execution. Examples of such objects include semaphores, message queues, signals, and pipes, as well as other types of objects. Each of these is discussed in detail in later chapters of this book.

The concept of concurrency and how an application is optimally decomposed into concurrent tasks is also discussed in more detail later in this book. For now, remember that the task object is the fundamental construct of most kernels. Tasks, along with task-management services, allow developers to design applications for concurrency to meet multiple time constraints and to address various design problems inherent to real-time embedded applications.

5. Define i) semaphores. ii) Message queue.iii) states.

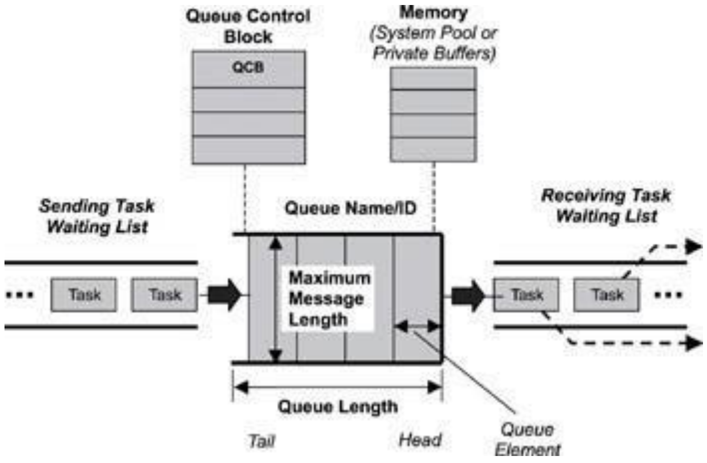
A *semaphore* (sometimes called a *semaphore token*) is a kernel object that one or more threads of execution can acquire or release for the purposes of synchronization or mutual exclusion.

When a semaphore is first created, the kernel assigns to it an associated semaphore control block (SCB), a unique ID, a value (binary or a count), and a task-waiting list, as shown in Figure 6.1.

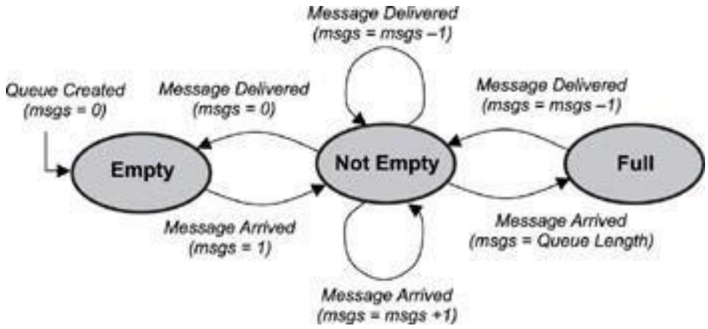


ii)**A message queue** is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchornize with data. A message queue is like a pipeline. It temporarily holds messages from a sender until the intended receiver is ready to read them. This temporary buffering decouples a sending and receiving task; that is, it frees the tasks from having to send and receive messages simultaneously.

As with semaphore introduced in Chapter 6, a message queue has several associated components that the kernel uses to manage the queue. When a message queue is first created, it is assigned an associated queue control block (QCB), a message queue name, a unique ID, memory buffers, a queue length, a maximum message length, and one or more task-waiting lists, as illustrated in Figure 7.1



States: As with other kernel objects, message queues follow the logic of a simple FSM, as shown in Figure 7.2 When a message queue is first created, the FSM is in the empty state. If a task attempts to receive messages from this message queue while the queue is empty, the task blocks and, if it chooses to, is held on the message queue's
task-waiting list, in either a FIFO or priority-based order.


Message Delivered
(msgs = msgs –1)
Queue Created
(msgs = 0)
Message Delivered
(msgs = 0)
Message Delivered
(msgs = msgs –1)
Empty
Not Empty
Full
Message Arrived
(msgs = 1)
Message Arrived
(msgs = Queue Length)
Message Arrived
(msgs = msgs +1)