Unit V - TASK COMMUNICATIONS

INTERPROCESS COMMUNICATION MECHANISMS

Processes often need to communicate with each other. ***Interprocess communication mechanisms*** are provided by the operating system as part of the processabstraction.

In general, a process can send a communication in one of two ways: blocking or nonblocking. After sending a blocking communication, the process goes into the waiting state until it receives a response. Nonblocking communication allows the process to continue execution after sending the communication. Both types of communication are useful.

There are two major styles of interprocess communication: shared memory and message passing.The two are logically equivalent—given one,you can build an interface that implements the other. However, some programs may be easier to write using one rather than the other. In addition,the hardware platform may make one easier to implement or more efficient than the other.

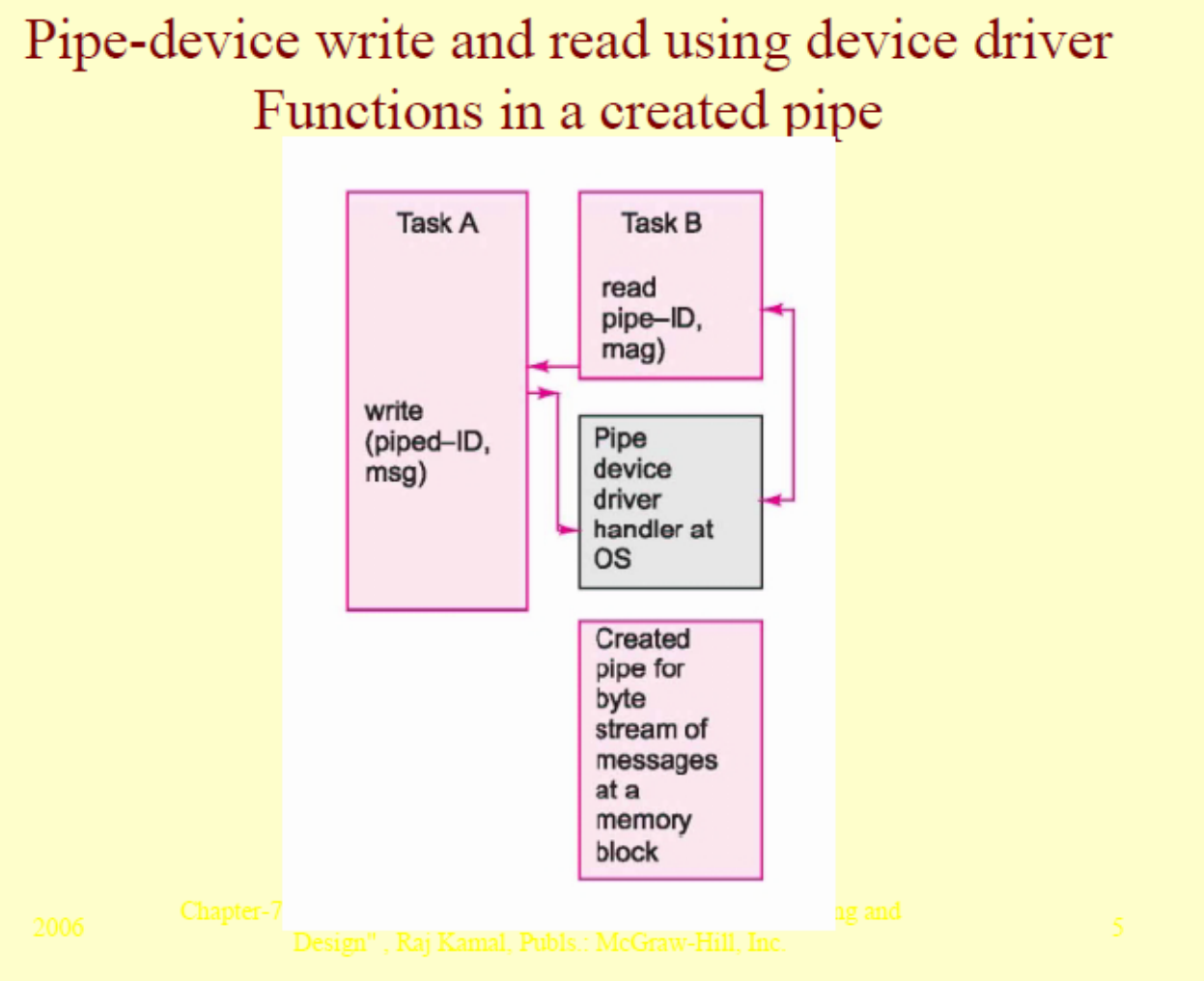
Pipe

• Pipe is a device used for the inter process communication

• Pipe has the functions create, connect and delete and functions similar to a device driver (open, write, read, close)

• A message-pipe ─ a device for inserting (writing) and deleting (reading) from that between two given inter-connected tasks or two sets of tasks.

• Writing and reading from a pipe is like using a C command fwrite with a file name to write into a named file, and C command fread with a file name to read into a named file.



Write and read using Pipe

1. One task using the function fwrite in a set of tasks can write *to* a pipe at the back pointer address, *pBACK.

2. One task using the function fwrite in a set of tasks can read from a pipe at the front pointer address, *Pfront

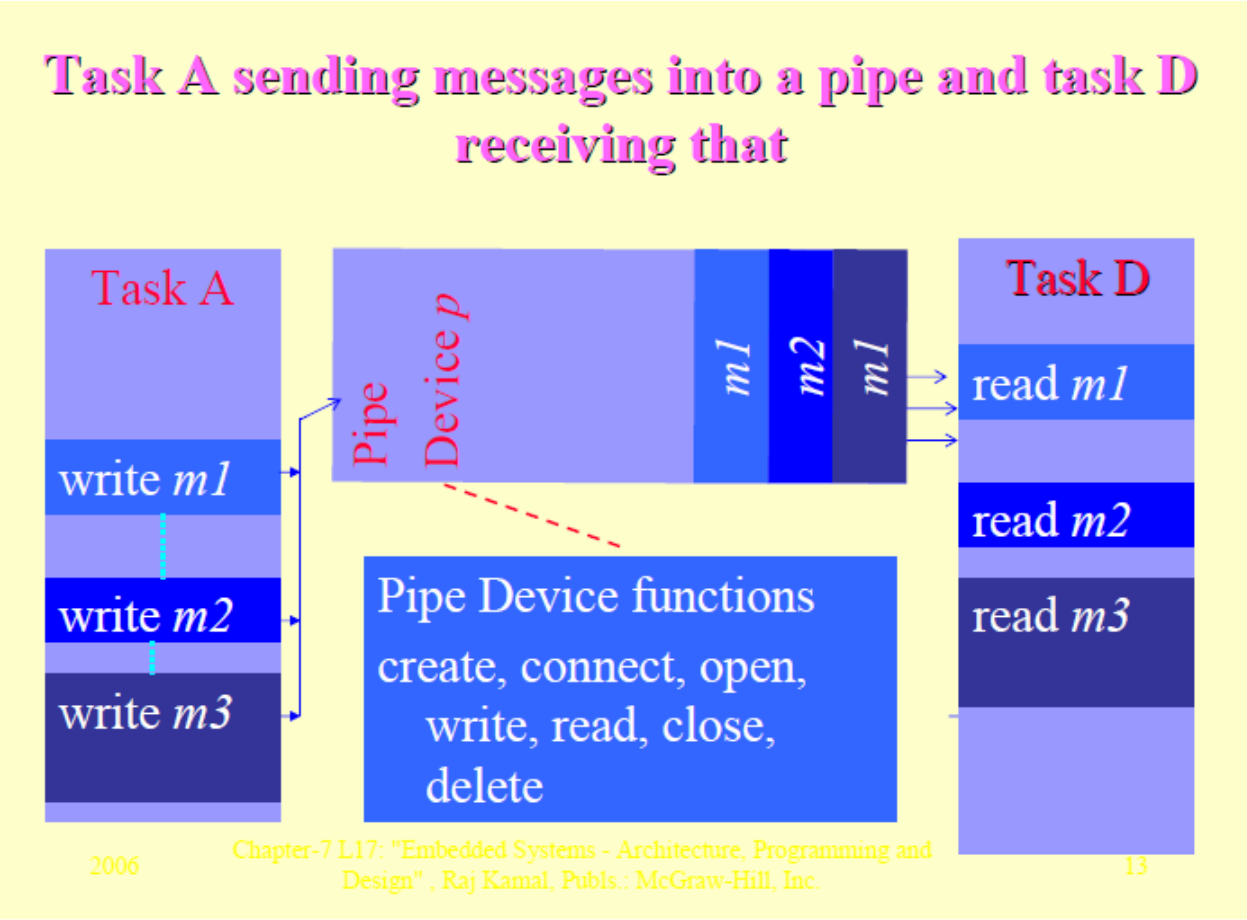
Pipe Messages

In a pipe there may be no fixed number of bytes per message with an initial pointer for the back and front and there may be a limiting final back pointer.

A pipe can therefore be limited and have a variable number of bytes per message between the initial and final pointers

Unidirectional feature in Pipe

Pipe is unidirectional. One thread or task inserts into it and other one deletes from it.



Shared Memory Communication

Figure illustrates how shared memory communication works in a bus-based system. Two components, such as a CPU and an I/O device, communicate through a shared memory location. The software on the CPU has been designed to know the address of the shared location;the shared location has also been loaded into the proper register of the I/O device. If, as in the figure, the CPU wants to send data to the device, it writes to the shared location.The I/O device then reads the data from that location. The read and write operations are standard and can be encapsulated in a procedural interface.

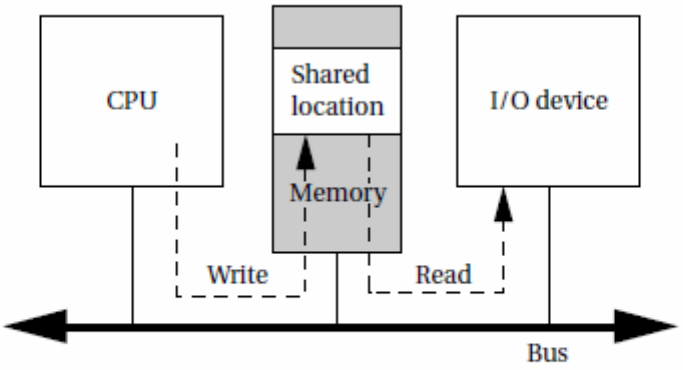


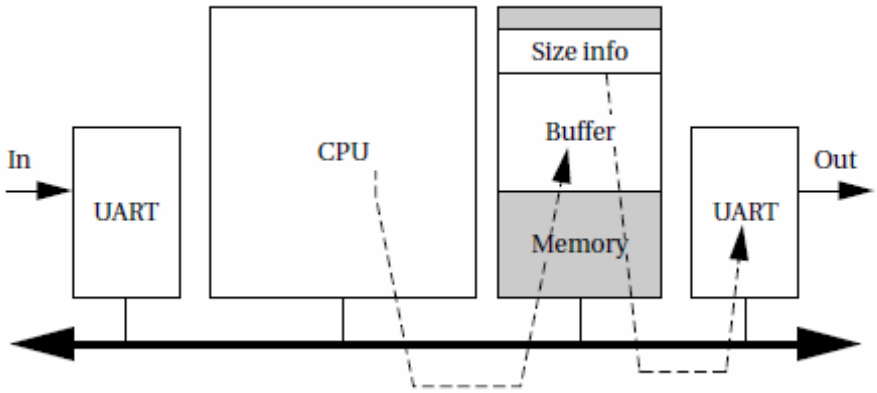
FIGURE 6.14

Shared memory communication implemented on a bus.

Example 6.7 describes the use of shared memory as a practical communication mechanism.

Example 6.7

Elastic buffers as shared memory



The text compressor provides a good example of a shared memory.

As shown below, the text compressor uses the CPU to compress incoming text, which is then sent on a serial line by a UART.

The input data arrive at a constant rate and are easy to manage. But because the output data are consumed at a variable rate, these data require an elastic buffer. The CPU and output

UART share a memory area—the CPU writes compressed characters into the buffer and the UART removes them as necessary to fill the serial line. Because the number of bits in the buffer changes constantly, the compression and transmission processes need additional size information. In this case, coordination is simple—the CPU writes at one end of the buffer and the UART reads at the other end. The only challenge is to make sure that the UART does not overrun the buffer.

As an application of shared memory,let us consider the situation in which the CPU and the I/O device want to communicate through a shared memory block. There must be a flag that tells the CPU when the data from the I/O device is ready. The flag, an additional shared data location, has a value of 0 when the data are not ready and 1 when the data are ready.The CPU, for example,would write the data,and then set the flag location to 1. If the flag is used only by the CPU, then the flag can be implemented using a standard memory write operation. If the same flag is used for bidirectional signaling between the CPU and the I/O device, care must be taken. Consider the following scenario:

1. CPU reads the flag location and sees that it is 0.

2. I/O device reads the flag location and sees that it is 0.

3. CPU sets the flag location to 1 and writes data to the shared location.

4. I/O device erroneously sets the flag to 1 and overwrites the data left by the CPU.

The above scenario is caused by a critical timing race between the two programs.

To avoid such problems, the microprocessor bus must support an atomic ***test-andset*** operation, which is available on a number of microprocessors. The test-and-set operation first reads a location and then sets it to a specified value. It returns the

result of the test. If the location was already set,then the additional set has no effect but the test-and-set instruction returns a false result. If the location was not set, the

instruction returns true and the location is in fact set. The bus supports this as an ***atomic*** operation that cannot be interrupted.

A test-and-set can be used to implement a ***semaphore***,which is a language-level synchronization construct. For the moment, let's assume that the system provides one semaphore that is used to guard access to a block of protected memory. Any process that wants to access the memory must use the semaphore to ensure that no other process is actively using it. As shown below,the semaphore names by tradition are P( ) to gain access to the protected memory and V( ) to release it.

/* some nonprotected operations here */

P(); /* wait for semaphore */

/* do protected work here */

V(); /* release semaphore */

The P( ) operation uses a test-and-set to repeatedly test a location that holds a lock on the memory block. The P( ) operation does not exit until the lock is available; once it is available, the test-and-set automatically sets the lock.

Once past,the P( ) operation, the process can work on the protected memory block. The V( ) operation resets the lock, allowing other processes access to the region by using the P( ) function.

Message Passing

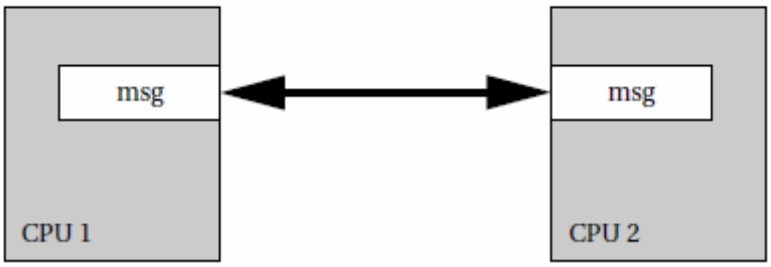


FIGURE 6.15
Message passing communication.

- Message passing communication complements the shared memory model.
- As shown in Figure each communicating entity has its own message send/receive unit.
- The message is not stored on the communications link, but rather at the senders/ receivers at the end points. In contrast,shared memory communication can be seen as a memory block used as a communication device,in which all the data are stored in the communication link/memory.
- Applications in which units operate relatively autonomously are natural candidates for message passing communication. For example, a home control system has one microcontroller per household device—lamp, thermostat, faucet, appliance, and so on.
- The devices must communicate relatively infrequently; furthermore, their physical separation is large enough that we would not naturally think of them as sharing a central pool of memory.
- Passing communication packets among the devices is a natural way to describe coordination between these devices. Message passing is the natural implementation of communication in many 8-bit microcontrollers that do not normally operate with external memory.

Signals

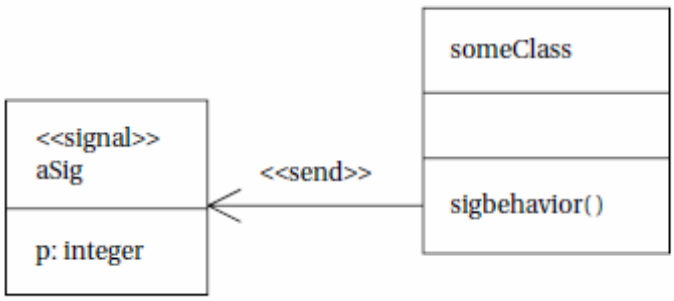


FIGURE 6.16
Use of a UML signal.

Another form of interprocess communication commonly used in Unix is the ***signal***.

A signal is simple because it does not pass data beyond the existence of the signal itself. A signal is analogous to an interrupt, but it is entirely a software creation.

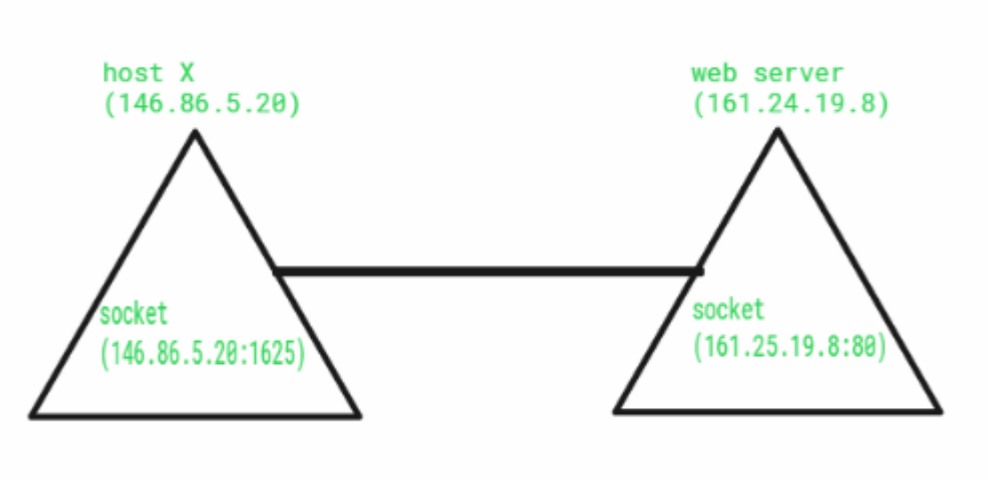
A signal is generated by a process and transmitted to another process by the operating system.

A UML signal is actually a generalization of the Unix signal. While a Unix signal

carries no parameters other than a condition code,a UML signal is an object.As such, it can carry parameters as object attributes. Figure 6.16 shows the use of a signal in UML. The *sigbehavior*( ) behavior of the class is responsible for throwing the signal,as indicated by<<*send*>>.The signal object is indicated by the<<*signal*>> stereotype.

Sockets

- A **socket** is one endpoint of a **two way** communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication take place.
- Like 'Pipe' is used to create pipes and sockets is created using **'socket'** system call. The socket provides bidirectional **FIFO** Communication facility over the network. A socket connecting to the network is created at each end of the communication. Each socket has a specific address. This address is composed of an IP address and a port number.
- Socket are generally employed in client server applications. The server creates a socket, attaches it to a network port addresses then waits for the client to contact it. The client creates a socket and then attempts to connect to the server socket. When the connection is established, transfer of data takes place.

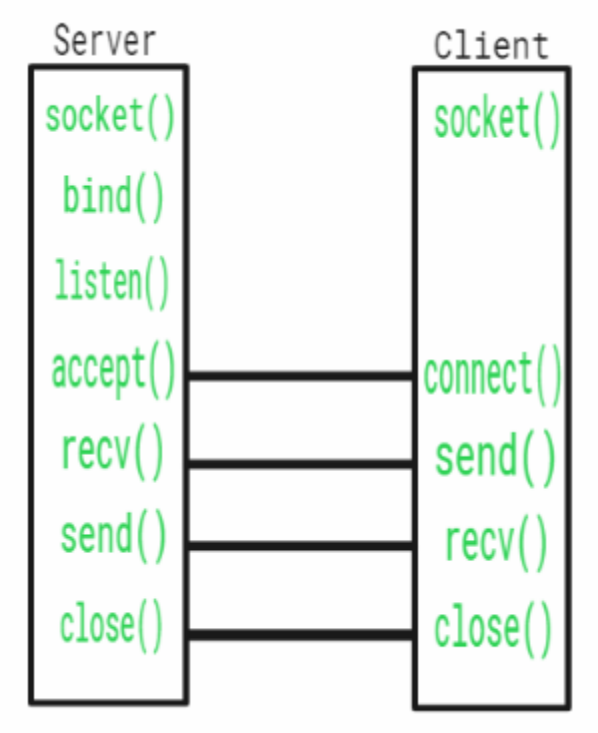


TypesofSockets:

There are two types of Sockets: the **datagram** socket and the **stream** socket.

1. **DatagramSocket:**
   This is a type of network which has connection less point for sending and receiving packets. It is similar to mailbox. The letters (data) posted into the box are collected and delivered (transmitted) to a letterbox (receiving socket).

2. **StreamSocket**
   In Computer operating system, a stream socket is type of interprocess communications socket or network socket which provides a connection-oriented, sequenced, and unique flow of data without record boundaries with well defined mechanisms for creating and destroying connections and for detecting errors. It is similar to phone. A connection is established between the phones (two ends) and a conversation (transfer of data) takes place.



Function Call	Description
Create()	To create a socket
Bind()	It's a socket identification like a telephone number to contact
Listen()	Ready to receive a connection
Connect()	Ready to act as a sender
Accept()	Confirmation, it is like accepting to receive a call from a sender
Write()	To send data
Read()	To receive data
Close()	To close a connection

Remote Procedure Call (RPC)

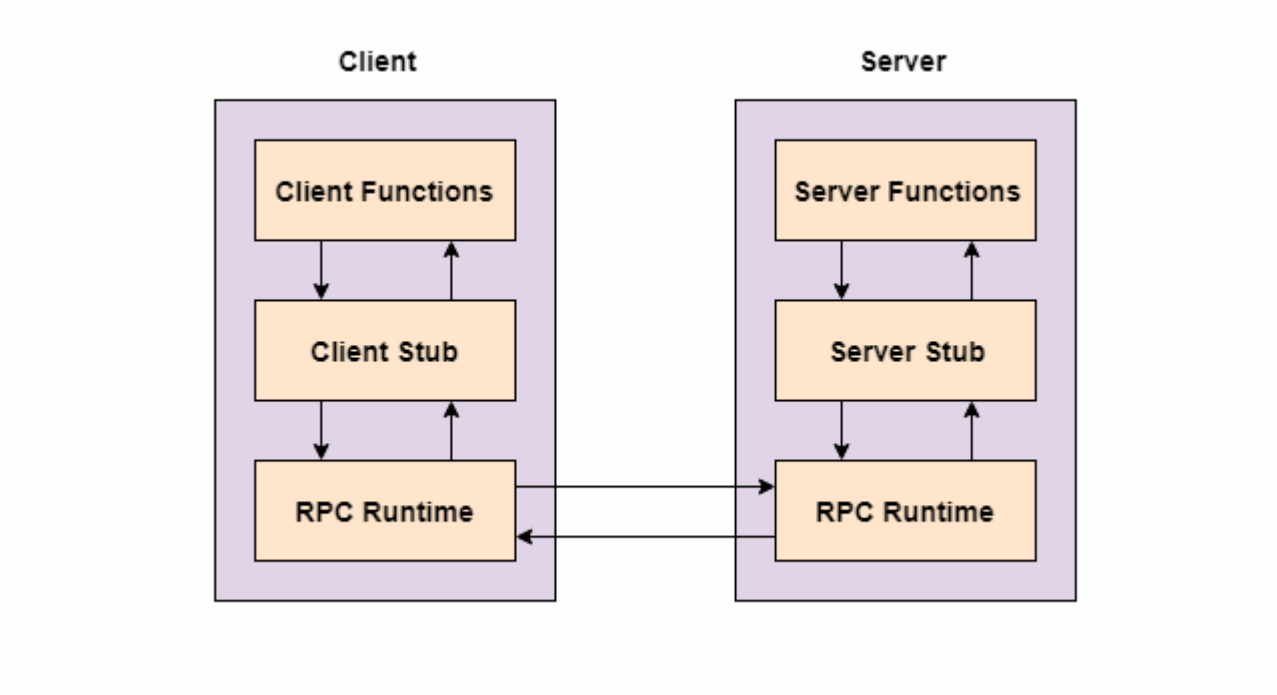
A remote procedure call is an interprocess communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call.

A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished.

The sequence of events in a remote procedure call are given as follows −

- The client stub is called by the client.
- The client stub makes a system call to send the message to the server and puts the parameters in the message.
- The message is sent from the client to the server by the client's operating system.
- The message is passed to the server stub by the server operating system.
- The parameters are removed from the message by the server stub.
- Then, the server procedure is called by the server stub.

A diagram that demonstrates this is as follows −



Advantages of Remote Procedure Call

Some of the advantages of RPC are as follows −

- Remote procedure calls support process oriented and thread oriented models.
- The internal message passing mechanism of RPC is hidden from the user.
- The effort to re-write and re-develop the code is minimum in remote procedure calls.
- Remote procedure calls can be used in distributed environment as well as the local environment.
- Many of the protocol layers are omitted by RPC to improve performance.

Disadvantages of Remote Procedure Call

Some of the disadvantages of RPC are as follows −

- The remote procedure call is a concept that can be implemented in different ways. It is not a standard.

- There is no flexibility in RPC for hardware architecture. It is only interaction based.
- There is an increase in costs because of remote procedure call.

Synchronization

Synchronization is classified into two categories:

resource synchronization and *activity synchronization*.

Resource synchronization determines whether access to a shared resource is safe, and, if not, when it will be safe.

Activity synchronization determines whether the execution of a multithreaded program has reached a certain state and, if it hasn't, how to wait for and be notified when this state is reached.

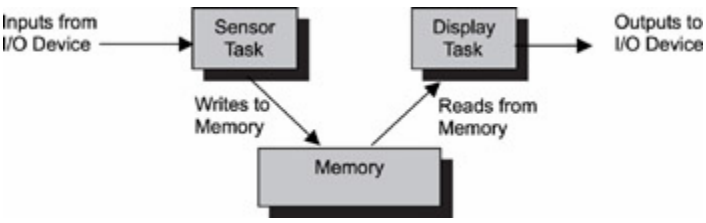
Resource Synchronization

Access by multiple tasks must be synchronized to maintain the integrity of a shared resource. This process is called *resource synchronization*, a term closely associated with critical sections and mutual exclusions.

Mutual exclusion is a provision by which only one task at a time can access a shared resource. A *critical section* is the section of code from which the shared resource is accessed.

As an example, consider two tasks trying to access shared memory. One task (the sensor task) periodically receives data from a sensor and writes the data to shared memory. Meanwhile, a second task (the display task) periodically reads from shared memory and sends the data to a display. The common design pattern of using shared memory is illustrated in figure.

.



Multiple tasks accessing shared memory.

Problems arise if access to the shared memory is not exclusive, and multiple tasks can simultaneously access it. For example, if the sensor task has not completed writing data to the shared memory area before the display task tries to display the data, the display would contain a mixture of data extracted at different times, leading to erroneous data interpretation.

The section of code in the sensor task that writes input data to the shared memory is a critical section of the sensor task. The section of code in the display task that reads data from the shared memory is a critical section of the display task. These two critical sections are called *competing critical sections* because they access the same shared resource.

A mutual exclusion algorithm ensures that one task's execution of a critical section is not interrupted by the competing critical sections of other concurrently executing tasks.

One way to synchronize access to shared resources is to use a client-server model, in which a central entity called a *resource server* is responsible for synchronization. Access requests are made to the resource server, which must grant permission to the requestor before the requestor can access the shared resource. The resource server determines the eligibility of the requestor based on pre-assigned rules or run-time heuristics.

While this model simplifies resource synchronization, the resource server is a bottleneck. Synchronization primitives, such as semaphores and mutexes allow developers to implement complex mutual exclusion algorithms. These algorithms in turn allow dynamic coordination among competing tasks without intervention from a third party.

Activity Synchronization

In general, a task must synchronize its activity with other tasks to execute a multithreaded program properly. *Activity synchronization* is also called *condition synchronization* or *sequence control* .

Activity synchronization ensures that the correct execution order among cooperating tasks is used. Activity synchronization can be either synchronous or asynchronous.

One representative of activity synchronization methods is *barrier synchronization* . For example, in embedded control systems, a complex computation can be divided and distributed among multiple tasks. Some parts of this complex computation are

I/O bound, other parts are CPU intensive, and still others are mainly floating-point operations that rely heavily on specialized floating-point coprocessor hardware. These partial results must be collected from the various tasks for the final calculation. The result determines what other partial computations each task is to perform next.

The point at which the partial results are collected and the duration of the final computation is a *barrier* . One task can finish its partial computation before other tasks complete theirs, but this task must wait for all other tasks to complete their computations before the task can continue.

Barrier synchronization comprises three actions:

- a task posts its arrival at the barrier,
- the task waits for other participating tasks to reach the barrier, and
- the task receives notification to proceed beyond the barrier.

As shown in Figure, a group of five tasks participates in barrier synchronization. Tasks in the group complete their partial execution and reach the barrier at various times; however, each task in the group must wait at the barrier until all other tasks have reached the barrier. The last task to reach the barrier (in this example, task T5) broadcasts a notification to the other tasks. All tasks cross the barrier at the same time (conceptually in a uniprocessor environment due to task scheduling. We say 'conceptually' because in a uniprocessor environment, only one task can execute at any given time. Even though all five tasks have crossed the barrier and may continue execution, the task with the highest priority will execute next.

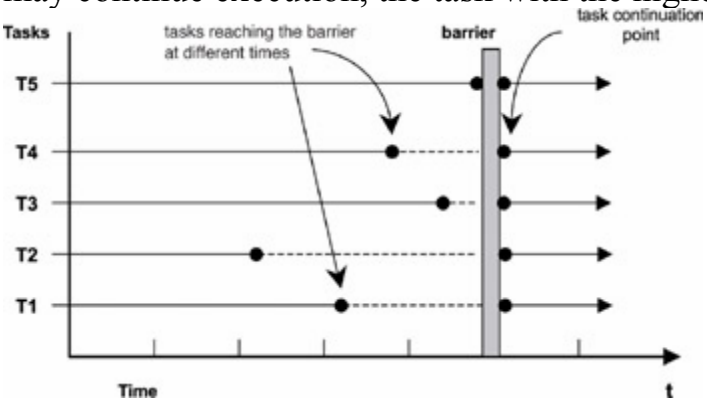


Figure: Visualization of barrier synchronization.

Another representative of activity synchronization mechanisms is *rendezvous synchronization* , which, as its name implies, is an execution point where two tasks meet. The main difference between the barrier and the rendezvous is that the

barrier allows activity synchronization among two or more tasks, while rendezvous synchronization is between two tasks.

In rendezvous synchronization, a synchronization and communication point called an *entry* is constructed as a function call. One task defines its entry and makes it public. Any task with knowledge of this entry can call it as an ordinary function call. The task that defines the entry accepts the call, executes it, and returns the results to the caller. The issuer of the entry call establishes a rendezvous with the task that defined the entry.

.

Two tasks can implement a simple rendezvous without data passing by using two binary semaphores, as shown in Figure

.

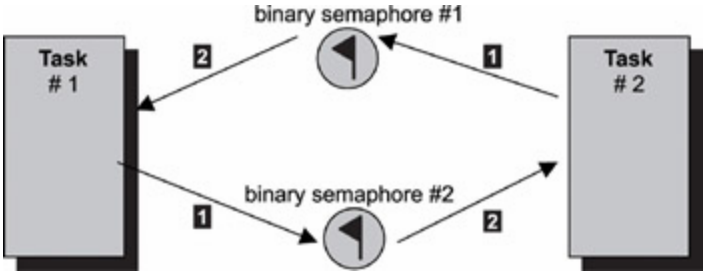


Figure : Simple rendezvous without data passing.

Both binary semaphores are initialized to 0 . When task #1 reaches the rendezvous, it gives semaphore #2, and then it gets on semaphore #1. When task #2 reaches the rendezvous, it gives semaphore #1, and then it gets on semaphore #2. Task #1 has to wait on semaphore #1 before task #2 arrives, and vice versa, thus achieving rendezvous synchronization.

Resource Synchronization Methods

Interrupt Locks

Interrupt locking (disabling system interrupts) is the method used to synchronize exclusive access to shared resources between tasks and ISRs. Some processor architecture designs allow for a fine-grained, interrupt-level lock, i.e., an interrupt lock level is specified so that asynchronous events at or below the level of the

disabled interrupt are blocked for the duration of the lock. Other processor architecture designs allow only coarse-grained locking, i.e., all system interrupts are disabled.

When interrupts are disabled at certain levels, even the kernel scheduler cannot run because the system becomes non-responsive to those external events that can trigger task re-scheduling. This process guarantees that the current task continues to execute until it voluntarily relinquishes control. As such, interrupt locking can also be used to synchronize access to shared resources between tasks.

Interrupt locking is simple to implement and involves only a few instructions. However, frequent use of interrupt locks can alter the overall system timing, with side effects including missed external events (resulting in data overflow) and clock drift (resulting in missed deadlines).

Interrupt locks, although the most powerful and the most effective synchronization method, can introduce indeterminism into the system when used indiscriminately. Therefore, the duration of interrupt locks should be short, and interrupt locks should be used only when necessary to guard a task-level critical region from interrupt activities.

A task that enabled interrupt locking must avoid blocking. The behavior of a task making a blocking call (such as acquiring a semaphore in blocking mode) while interrupts are disabled is dependent on the RTOS implementation. Some RTOSes block the calling task and then re-enable the system interrupts. The kernel disables interrupts again on behalf of the task after the task is ready to be unblocked. The system can hang forever in RTOSes that do not support this feature.

Preemption Locks

Preemption locking (disabling the kernel scheduler) is another method used in resource synchronization. Many RTOS kernels support priority-based, preemptive task scheduling. A task disables the kernel preemption when it enters its critical section and re-enables the preemption when finished. The executing task cannot be preempted while the preemption lock is in effect.

On the surface, preemption locking appears to be more acceptable than interrupt locking. Closer examination reveals that preemption locking introduces the possibility for priority inversion.

Even though interrupts are enabled while preemption locking is in effect, actual servicing of the event is usually delayed to a dedicated task outside the context of the ISR. The ISR must notify that task that such an event has occurred.

This dedicated task usually executes at a high priority. This higher priority task, however, cannot run while another task is inside a critical region that a preemption lock is guarding.

The problem with preemption locking is that higher priority tasks cannot execute, even when they are totally unrelated to the critical section that the preemption lock is guarding. This process can introduce indeterminism in a similar manner to that caused by the interrupt lock. This indeterminism is unacceptable to many systems requiring consistent real-time response.

For example, consider two medium-priority tasks that share a critical section and that use preemption locking as the synchronization primitive. An unrelated print server daemon task runs at a much higher priority; however, the printer daemon cannot send a command to the printer to eject one page and feed the next while either of the medium tasks is inside the critical section. This issue results in garbled output or output mixed from multiple print jobs.

The benefit of preemption locking is that it allows the accumulation of asynchronous events instead of deleting them. The I/O device is maintained in a consistent state because its ISR can execute. Unlike interrupt locking, preemption locking can be expensive, depending on its implementation.

In the majority of RTOSes when a task makes a blocking call while preemption is disabled, another task is scheduled to run, and the scheduler disables preemption after the original task is ready to resume execution.

Critical Section

The critical section of a task is a section of code that accesses a shared resource. A competing critical section is a section of code in another task that accesses the same resource.

If these tasks do not have real-time deadlines and guarding the critical section is used only to ensure exclusive access to the shared resource without side effects, then the duration of the critical section is not important.

Imagine that a system has two tasks: one that performs some calculations and stores the results in a shared variable and another that reads that shared variable and displays its value. Using a chosen mutual exclusion algorithm to guard the critical section ensures that each task has exclusive access to the shared variable. These tasks do not have real-time requirements, and the only constraint placed on these two tasks is that the write operation precedes the read operation on the shared variable.

If another task without a competing critical section exists in the system but does have real-time deadlines to meet, the task must be allowed to interrupt either of the other two tasks, regardless of whether the task to be interrupted is in its critical section, in order to guarantee overall system correctness. Therefore, in this particular example, the duration of the critical sections of the first two tasks can be long, and higher priority task should be allowed to interrupt.

If the first two tasks have real-time deadlines and the time needed to complete their associated critical sections impacts whether the tasks meet their deadlines, this critical section should run to completion without interruption. The preemption lock becomes useful in this situation.

Therefore, it is important to evaluate the criticality of the critical section and the overall system impact before deciding on which mutual exclusion algorithm to use for guarding a critical section. The solution to the mutual exclusion problem should satisfy the following conditions:

- only one task can enter its critical section at any given time,

- fair access to the shared resource by multiple competing tasks is provided, and
- one task executing its critical section must not prevent another task executing a non-competing critical section.

Common Practical Design Patterns

This section presents a set of common inter-tasks synchronization and communication patterns designed from real-life scenarios. These design patterns are ready to be used in real-world embedded designs.

In these design patterns, the operation of event register manipulation is considered an atomic operation. The numberings shown in these design patterns indicate the execution orders.

Synchronous Activity Synchronization

Multiple ways of implementing synchronous activity synchronization are available, including:

- task-to-task synchronization using binary semaphores,
- ISR-to-task synchronization using binary semaphores,
- task-to-task synchronization using event registers,
- ISR-to-task synchronization using event registers,
- ISR-to-task synchronization using counting semaphores, and
- simple rendezvous with data passing.

Task-to-Task Synchronization Using Binary Semaphores

In this design pattern, two tasks synchronize their activities using a binary semaphore, as shown in Figure. The initial value of the binary semaphore is 0. Task #2 has to wait for task #1 to reach an execution point, at which time, task #1 signals to task #2 its arrival at the execution point by giving the semaphore and changing the value of the binary semaphore to 1. At this point, depending on their execution priorities, task #2 can run if it has higher priority. The value of the

binary semaphore is reset to 0 after the synchronization. In this design pattern, task #2 has execution dependency on task #1.

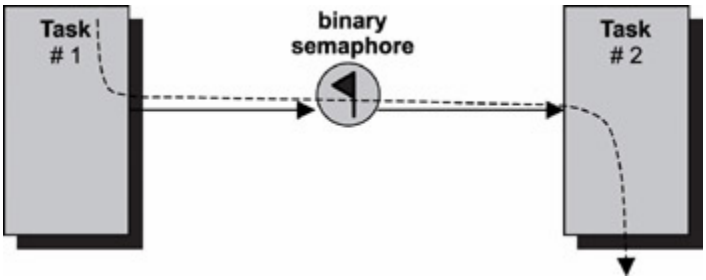


Figure Task-to-task synchronization using binary semaphores.

ISR-to-Task Synchronization Using Binary Semaphores

In this design pattern, a task and an ISR synchronize their activities using a binary semaphore, as shown in Figure. The initial value of the binary semaphore is 0. The task has to wait for the ISR to signal the occurrence of an asynchronous event. When the event occurs and the associated ISR runs, it signals to the task by giving the semaphore and changing the value of the binary semaphore to 1. The ISR runs to completion before the task gets the chance to resume execution. The value of the binary semaphore is reset to 0 after the task resumes execution.

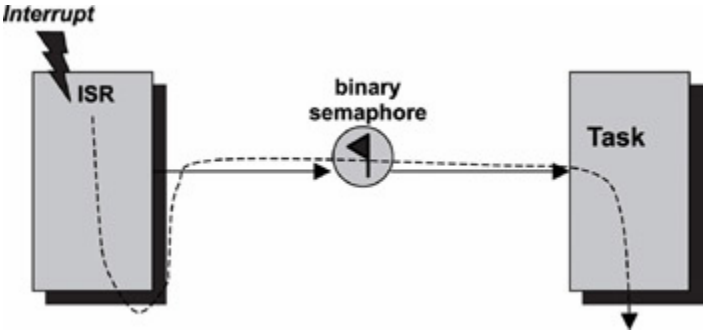


Figure ISR-to-task synchronization using binary semaphores.

Task-to-Task Synchronization Using Event Registers

In this design pattern, two tasks synchronize their activities using an event register, as shown in Figure. The tasks agree on a bit location in the event register for signaling. In this example, the bit location is the first bit. The initial value of the event bit is 0. Task #2 has to wait for task #1 to reach an execution point. Task #1 signals to task #2 its arrival at that point by setting the event bit to 1. At this point,

depending on execution priority, task #2 can run if it has higher priority. The value of the event bit is reset to 0 after synchronization.

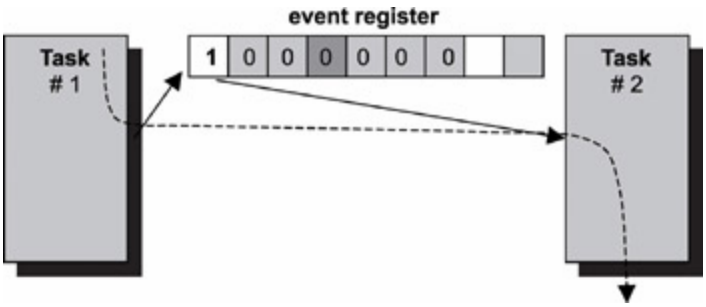


Figure: Task-to-task synchronization using event registers.

ISR-to-Task Synchronization Using Event Registers

In this design pattern, a task and an ISR synchronize their activities using an event register, as shown in Figure. The task and the ISR agree on an event bit location for signaling. In this example, the bit location is the first bit. The initial value of the event bit is 0. The task has to wait for the ISR to signal the occurrence of an asynchronous event. When the event occurs and the associated ISR runs, it signals to the task by changing the event bit to 1. The ISR runs to completion before the task gets the chance to resume execution. The value of the event bit is reset to 0 after the task resume execution.

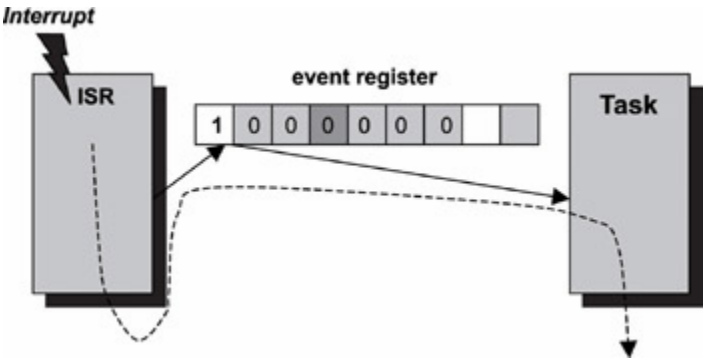


Figure ISR-to-task synchronization using event registers.

ISR-to-Task Synchronization Using Counting Semaphores

In previous figures, multiple occurrences of the same event cannot accumulate. A counting semaphore, however, is used in Figure to accumulate event occurrences and for task signaling. The value of the counting semaphore increments by one each time the ISR gives the semaphore. Similarly, its value is decremented by one

each time the task gets the semaphore. The task runs as long as the counting semaphore is non-zero.

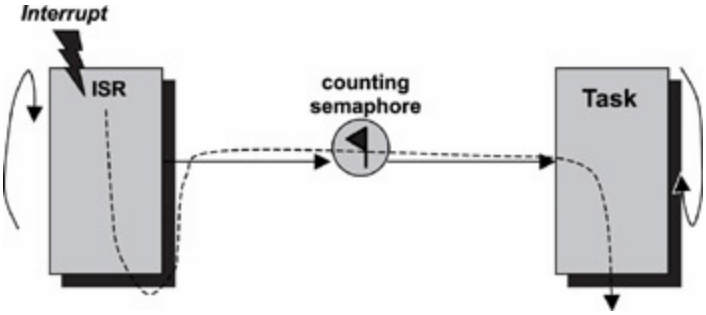


Figure 15.10: ISR-to-task synchronization using counting semaphores.

Simple Rendezvous with Data Passing

Two tasks can implement a simple rendezvous and can exchange data at the rendezvous point using two message queues, as shown in Figure. Each message queue can hold a maximum of one message. Both message queues are initially empty. When task #1 reaches the rendezvous, it puts data into message queue #2 and waits for a message to arrive on message queue #1. When task #2 reaches the rendezvous, it puts data into message queue #1 and waits for data to arrive on message queue #2. Task #1 has to wait on message queue #1 before task #2 arrives, and vice versa, thus achieving rendezvous synchronization with data passing.

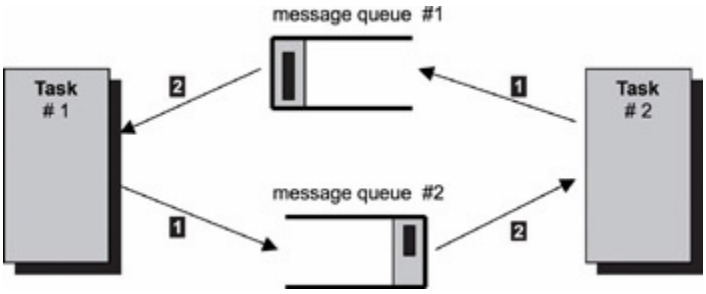


Figure: Task-to-task rendezvous using two message queues.

Asynchronous Event Notification Using Signals

One task can synchronize with another task in urgent mode using the signal facility. The signaled task processes the event notification asynchronously. In Figure, a task generates a signal to another task. The receiving task diverts from its normal execution path and executes its asynchronous signal routine.

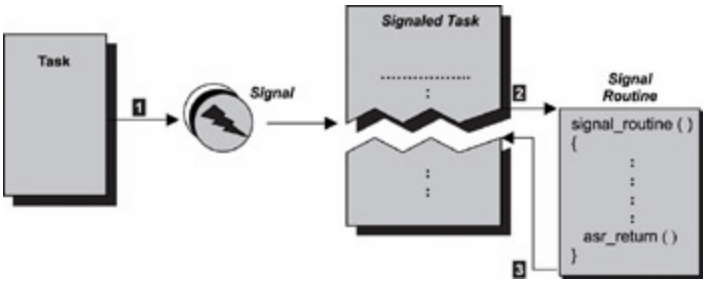


Figure Using signals for urgent data communication.

Resource Synchronization

Multiple ways of accomplishing resource synchronization are available. These methods include accessing shared memory with mutexes, interrupt locks, or preemption locks and sharing multiple instances of resources using counting semaphores and mutexes.

Shared Memory with Mutexes

In this design pattern, task #1 and task #2 access shared memory using a mutex for synchronization. Each task must first acquire the mutex before accessing the shared memory. The task blocks if the mutex is already locked, indicating that another task is accessing the shared memory. The task releases the mutex after it completes its operation on the shared memory. Figure shows the order of execution with respect to each task.

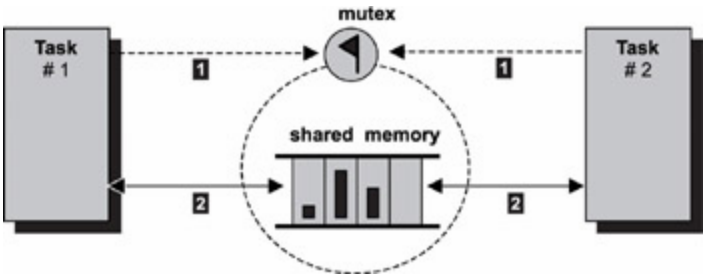


Figure : Task-to-task resource synchronization-shared memory guarded by mutex.

Shared Memory with Interrupt Locks

In this design pattern, the ISR transfers data to the task using shared memory, as shown in Figure. The ISR puts data into the shared memory, and the task removes data from the shared memory and subsequently processes it. The interrupt lock is used for synchronizing access to the shared memory. The task must acquire and release the interrupt lock to avoid the interrupt disrupting its execution. The ISR

does not need to be aware of the existence of the interrupt lock unless nested interrupts are supported (i.e., interrupts are enabled while an ISR executes) and multiple ISRs can access the data.

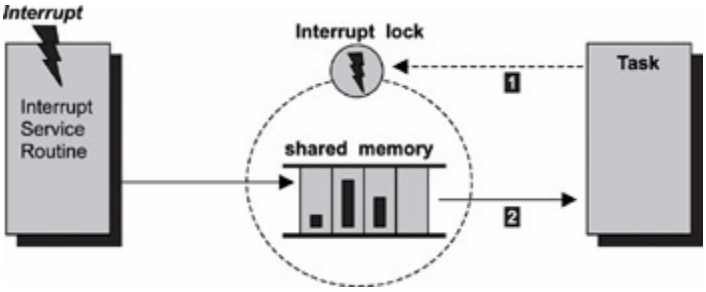


Figure ISR-to-task resource synchronization- shared memory guarded by interrupt lock.

Shared Memory with Preemption Locks

In this design pattern, two tasks transfer data to each other using shared memory, as shown in Figure. Each task is responsible for disabling preemption before accessing the shared memory. Unlike using a binary semaphore or a mutex lock, no waiting is involved when using a preemption lock for synchronization.

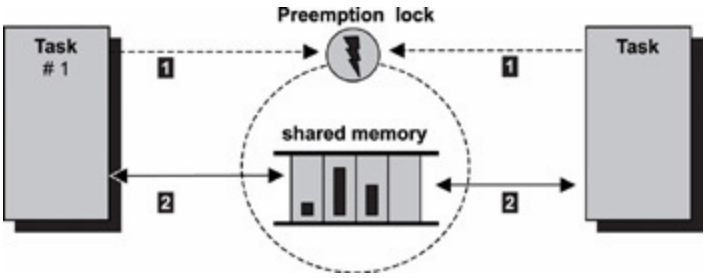


Figure: Task-to-task resource synchronization-shared memory guarded by preemption lock.

Sharing Multiple Instances of Resources Using Counting Semaphores and Mutexes

Figure depicts a typical scenario where *N* tasks share *M* instances of a single resource type, for example, *M* printers. The counting semaphore tracks the number of available resource instances at any given time. The counting semaphore is initialized with the value *M*. Each task must acquire the counting semaphore before accessing the shared resource. By acquiring the counting semaphore, the task effectively reserves an instance of the resource. Having the counting semaphore

alone is insufficient. Typically, a control structure associated with the resource instances is used. The control structure maintains information such as which resource instances are in use and which are available for allocation. The control information is updated each time a resource instance is either allocated to or released by a task. A mutex is deployed to guarantee that each task has exclusive access to the control structure. Therefore, after a task successfully acquires the counting semaphore, the task must acquire the mutex before the task can either allocate or free an instance.

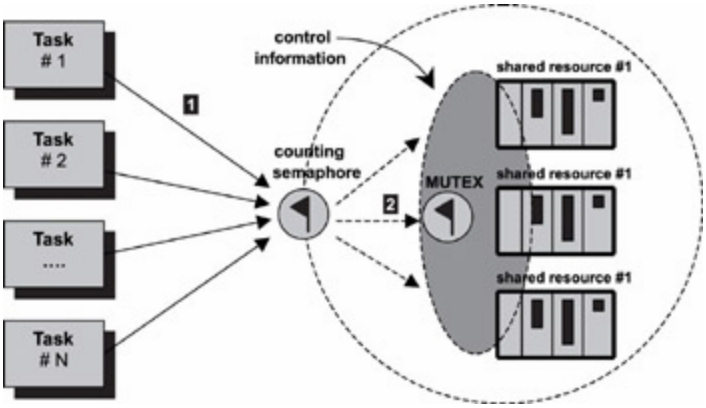


Figure: Sharing multiple instances of resources using counting semaphores and mutexes.

How to chose RTOS:

The decision of an RTOS for an embedded design is very critical.

A lot of factors need to be analyzed carefully before making a decision on the selection of an RTOS.

These factors can be either

1. Functional

2. Non-functional requirements.

1. Functional Requirements:

1. Processor support: • The RTOS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like FLASH.

It is not necessary that all RTOS's support all kinds of processor architectures.

It is essential to ensure the processor support by the RTOS

2. Memory Requirements:

OS also requires working memory RAM for loading the OS service.

Since embedded systems are memory constrained, it is essential to evaluate the minimal RAM and ROM requirements for the OS under consideration.

3. Real-Time Capabilities:

It is not mandatory that the OS for all embedded systems need to be Real- Time and all embedded OS's are 'Real-Time' in behavior.

The Task/process scheduling policies plays an important role in the Real- Time behavior of an OS.

3. Kernel and Interrupt Latency:

The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency.

For an embedded system whose response requirements are high, this latency should be minimal.

5. Inter process Communication (IPC) and Task Synchronization: The implementation of IPC and Synchronization is OS kernel dependent.

6. Modularization Support:

Most of the OS's provide a bunch of features.

It is very useful if the OS supports modularization where in which the developer can choose the essential modules and re-compile the OS image for functioning.

7. Support for Networking and Communication:

The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking.

Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.

8. Development Language Support:

Certain OS's include the run time libraries required for running applications written in languages like JAVA and C++.

The OS may include these components as built-in component, if not , check the availability of the same from a third party.

2. Non-Functional Requirements:

1. Custom Developed or Off the Shelf:

It is possible to go for the complete development of an OS suiting the embedded system needs or use an off the shelf, readily available OS.

It may be possible to build the required features by customizing an open source OS.

The decision on which to select is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.

2. Cost:

The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.

3. Development and Debugging tools Availability:

The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design.

Certain OS's may be superior in performance, but the availability of tools for supporting the development may be limited.

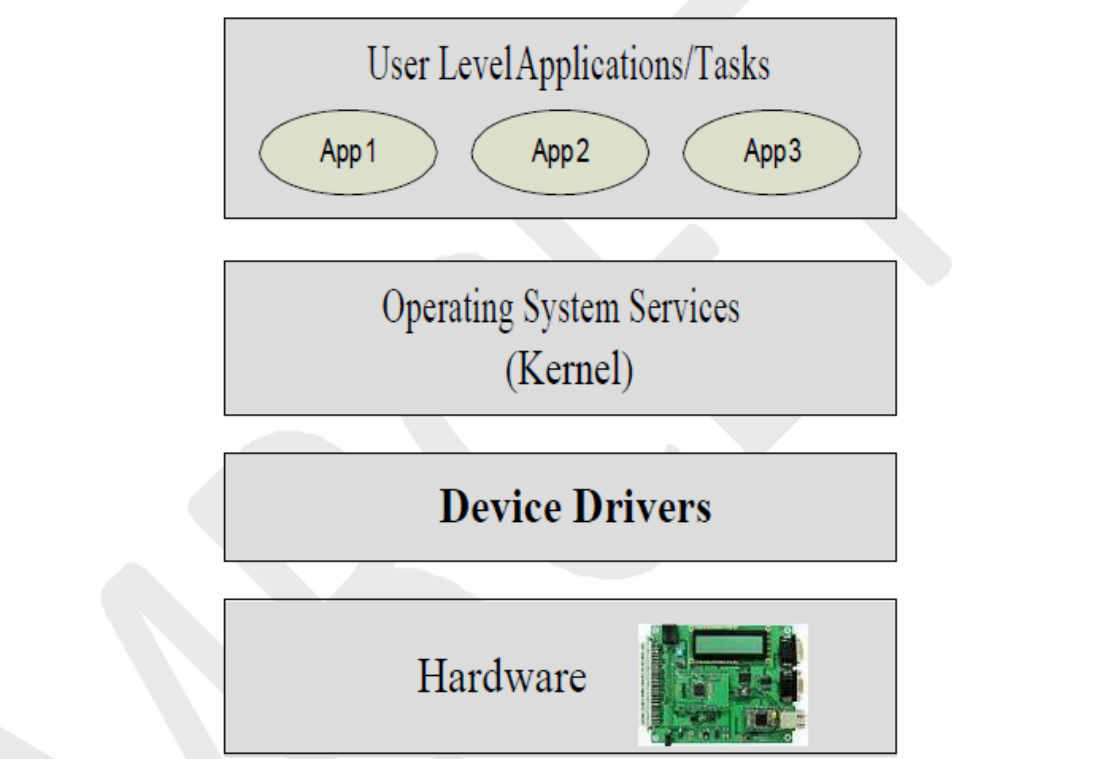
4. Ease of Use:

How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.

5. After Sales:

For a commercial embedded RTOS, after sales in the form of e-mail, on-call services etc. for bug fixes, critical patch updates and support for production issues etc. should be analyzed thoroughly.

Device Drivers:



- Device driver is a piece of software that acts as a bridge between the operating system and the hardware

- The user applications talk to the OS kernel for all necessary information exchange including communication with the hardware peripherals

- The architecture of the OS kernel will not allow direct device access from the user application
- All the device related access should flow through the OS kernel and the OS kernel routes it to the concerned hardware peripheral
- OS Provides interfaces in the form of Application Programming Interfaces (APIs) for accessing the hardware
- The device driver abstracts the hardware from user applications

- Device drivers are responsible for initiating and managing the communication with the hardware peripherals
- Drivers which comes as part of the Operating system image is known as 'built-in drivers' or 'onboard' drivers. Eg. NAND FLASH driver
- Drivers which needs to be installed on the fly for communicating with add-on devices are known as 'Installable drivers'
- For installable drivers, the driver is loaded on a need basis when the device is present and it is unloaded when the device is removed/detached
- The 'Device Manager service of the OS kernel is responsible for loading and unloading the driver, managing the driver etc.
- The underlying implementation of device driver is OS kernel dependent
- The driver communicates with the kernel is dependent on the OS structure and implementation.
- Device drivers can run on either user space or kernel space

- Device drivers which run in user space are known as *user mode drivers* and the drivers which run in kernel space are known as *kernel mode drivers*
- User mode drivers are safer than kernel mode drivers
- If an error or exception occurs in a user mode driver, it won't affect the services of the kernel
- If an exception occurs in the kernel mode driver, it may lead to the kernel crash
- The way how a device driver is written and how the interrupts are handled in it are Operating system and target hardware specific.
- The device driver implements the following:
    - Device (Hardware) Initialization and Interrupt configuration
    - Interrupt handling and processing
    - Client interfacing (Interfacing with user applications)

- The basic Interrupt configuration involves the following.
    - Set the interrupt type (Edge Triggered (Rising/Falling) or Level Triggered (Low or High)), enable the interrupts and set the interrupt priorities.
    - The processor identifies an interrupt through IRQ.
    - IRQs are generated by the Interrupt Controller.
    - Register an Interrupt Service Routine (ISR) with an Interrupt Request (IRQ).

- When an interrupt occurs, depending on its priority, it is serviced and the corresponding ISR is invoked
- The processing part of an interrupt is handled in an ISR
- The whole interrupt processing can be done by the ISR itself or by invoking an Interrupt Service Thread (IST)
- The IST performs interrupt processing on behalf of the IS.