MODULE 5

RTOS & IDE FOR EMBEDDED SYSTEM DESIGN

Prepared by:
Mr.Chetan.R, Sr.Asst. Professor

1. With a neat diagram, explain operating system architecture

- ➢ The operating system acts as a *bridge between the user applications/tasks and the underlying system resources* through a set of system functionalities and services.
- ➢ The OS manages the system resources and makes them available to the user applications/tasks on a need basis.
- ➢ A normal computing system is a collection ofdifferent I/O subsystems, working, and storage memory.

❖ The primary functions of an operating system is
- Make the system convenient to use
- Organise and manage the system resources efficiently and correctly

Figure 10.1 gives an insight into the basic components of an operating system and their interfaces with rest of the world.

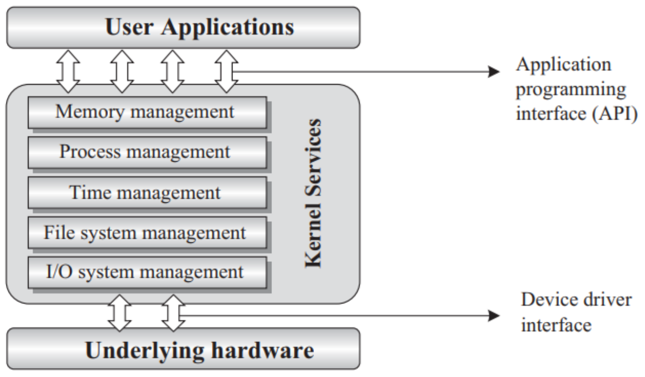


Fig. 10.1 The Operating System Architecture

The Kernel

- ➢ The kernel is the *core of the operating system* and is responsible for managing the system resources and the communication among the hardware and other system services.
- ➢ Kernel acts as the abstraction layer betweensystem resources and user applications.
- ➢ Kernel contains a set of system libraries and services.

For a general purpose OS, the kernel contains different services for handling the following.

Process Management: It includes setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, scheduling and managing the execution of the process, setting up and managing the Process Control Block (PCB), Inter Process Communication and synchronisation, process termination/ deletion, etc.

Primary Memory Management: The term primary memory refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored.
The Memory Management Unit (MMU) of the kernel is responsible for
- Keeping track of which part of the memory area is currently used by which process
- Allocating and De-allocating memory space on a need basis (Dynamic memory allocation).

File System Management: File is a collection of related information. A file could be a program (source code or executable), text files, image files, word documents, audio/video files, etc. Each of these files differ in the kind of information they hold and the way in which the information is stored. The file operation is a useful service provided by the OS.

The file system management service of Kernel is responsible for

- The creation, deletion and alteration of files
- Creation, deletion and alteration of directories
- Saving of files in the secondary storage memory (e.g. Hard disk storage)
- Providing automatic allocation of file space based on the amount of free space available
- Providing a flexible naming convention for the files

I/O System (Device) Management: Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system. In a well-structured OS, the direct accessing of I/O devices are not allowed and the access to them are provided through a set of Application ProgrammingInterfaces (APIs) exposed by the kernel.

The kernel maintains a list of all the I/O devices of the system. This list may be available in advance, at the time of building the kernel. Some kernels, dynamically updates the list of available devices as and when a new device is installed (e.g. Windows NT kernel keeps the list updatedwhen a new plug '*n*' play USB device is attached to the system).

The service 'Device Manager' of the kernel is responsible for handling all I/O device related operations. The kernel talks to the I/O device through a set of low-level systems calls, which are implemented in a service, called device drivers.

The device drivers are specific to a device or a class of devices. The Device Manager is responsible for

- Loading and unloading of device drivers
- Exchanging information and the system specific control signals to and from the device

Secondary Storage Management The secondary storage management deals with managing the secondary storage memory devices, if any, connected to the system. Secondary memory is used as backup medium for programs and data since the main memory is volatile. In most of the systems, the secondary storage is kept indisks (Hard Disk).

The secondary storage management service of kernel deals with

- Disk storage allocation
- Disk scheduling (Time interval at which the disk is activated to backup data)
- Free Disk space management

Protection Systems Most of the modern operating systems are designed in such a way to support multiple users with different levels of access permissions (e.g. Windows 10 with user permissions like 'Administrator', 'Standard', 'Restricted', etc.).

Protection deals with implementing the security policies to restrict the access to both user and system resources by different applications or processes or users. In multiuser supported operating systems, one user may not be allowed to view or modify the whole/portions of another user's data or profile details. In addition, some application may not be granted with permission to make use of some of the system resources. This kind of protection is provided by the protection services running within the kernel.

Interrupt Handler Kernel provides handler mechanism for all external/internal interrupts generated by the system. These are some of the important services offered by the kernel of an operating system. It does not mean that a kernel contains no more than components/services explained above.

Depending on the type of the operating system, a kernel may contain lesser number of components/services or more number of components/ services. In addition to the components/services listed above, many operating systems offer a number of add- on system components/services to the kernel. Network communication, network management, user-interface graphics, timer services (delays, timeouts, etc.), error handler, database management, etc. are examples for such components/services.

Kernel exposes the interface to the various kernel applications/services, hosted by kernel, to the user applications through a set of standard Application Programming Interfaces (APIs). User applications can avail these API calls to access the various kernel application/services.

2. Differentiate between hard real time and soft real time operating system with an example for each.

Hard Real-Time:

- Real-Time Operating Systems that strictly adhere to the timing constraints for a task is referred as '*Hard Real-Time*' systems.
- A Hard Real-Time system must meet the deadlines for a task without any slippage.
- Missing any deadline may produce catastrophic results for Hard Real-Time Systems, including permanent data lose and irrecoverable damages to the system/users.
- A system can have several such tasks and the key to their correct operation lies in scheduling them so that they meet their time constraints.
- Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples for Hard Real-Time Systems.
  - ➢ The Air bag control system should be into action and deploy the air bags when the vehicle meets a severe accident. Ideally speaking, the time for triggering the air bag deployment task, when an accident is sensed by the Air bag control system, should be zero and the air bags should be deployed exactly within the time frame, which is predefined for the air bag deployment task.

Soft Real-Time:

- Real-Time Operating System that does not guarantee meeting deadlines, but offer the best effort to meet the deadline are referred as '*Soft Real-Time*' systems.
- Missing deadlines for tasks are acceptable for a Soft Real-time system if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS).
- Automatic TellerMachine (ATM) is a typical example for Soft-Real-Time System.
  - ➢ If the ATM takes a few seconds more thanthe ideal operation time, nothing fatal happens.
  - ➢ An audio-video playback system is another example for SoftReal-Time system.

3. Define Task, Process and Threads

The term '*task*': It is defined as the program in execution and the related information maintained by the operating system for the program. Task is also known as '*Job*'. A program or part of it in execution is also called a '*Process*'. The terms '*Task*', '*Job*' and '*Process*' refer to the same entity in the operating system and most often they are used interchangeably.

A '*Process*' is a program, or part of it, in execution. Process is also known as an instance of a program in execution. Multiple instances of the same program can execute simultaneously. A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange, etc. A process is sequential in execution.

A thread is the primitive that can execute code. A *thread* is a single sequential flow of control within a process. '*Thread*' is also known as lightweight process. A process can have many threads of execution. Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area. Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack.

4. Explain the process structure, process states and state transistions

- The concept of '*Process*' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilisation of the CPU and other system resources.
- Concurrent execution is achieved through the sharing of CPU among the processes.
- A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process. This can be visualisedas shown in Fig. 10.4.
- A process which inherits all the properties of the CPU can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor. When the process gets its turn, its registers and the program counter register becomes mapped to the physical registers of the CPU.

- From a memory perspective, the memory occupied by the *process* is segregated into three regions, namely, Stack memory, Data memory and Code memory (Fig. 10.5).

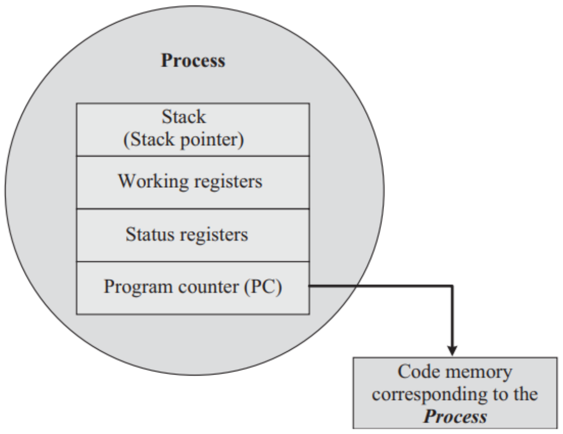


Fig. 10.4 Structure of a Process

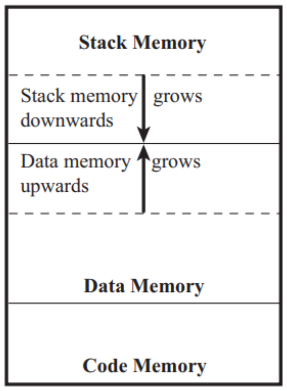


Fig. 10.5 Memory organisation of a Process

- ❖ The 'Stack' memory holds all temporary data such as variables local to the process.
- ❖ Data memory holds all global data for the process.
- ❖ The code memory contains the program code (instructions) correspondingto the process.

Process States and State Transition

- The creation of a process to its termination is nota single step operation.
- The process traverses through a series of states during its transition from the newly created state to the terminated state.
- The cycle through which a process changes its state from '*newly created*' to '*execution completed*' is known as '*Process Life Cycle*'.
- The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to do next. Figure 10.6 representsthe various states associated with a process.

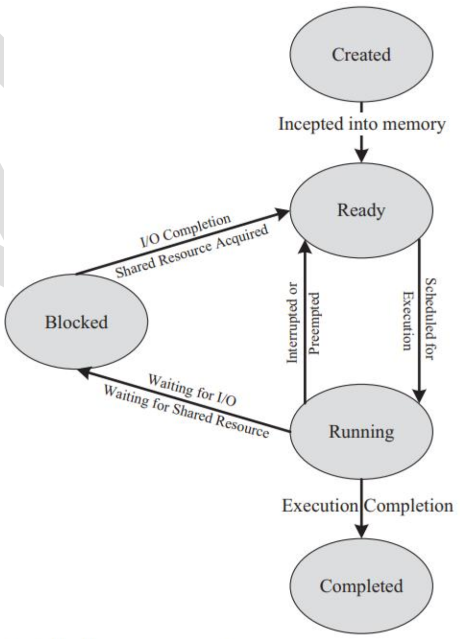


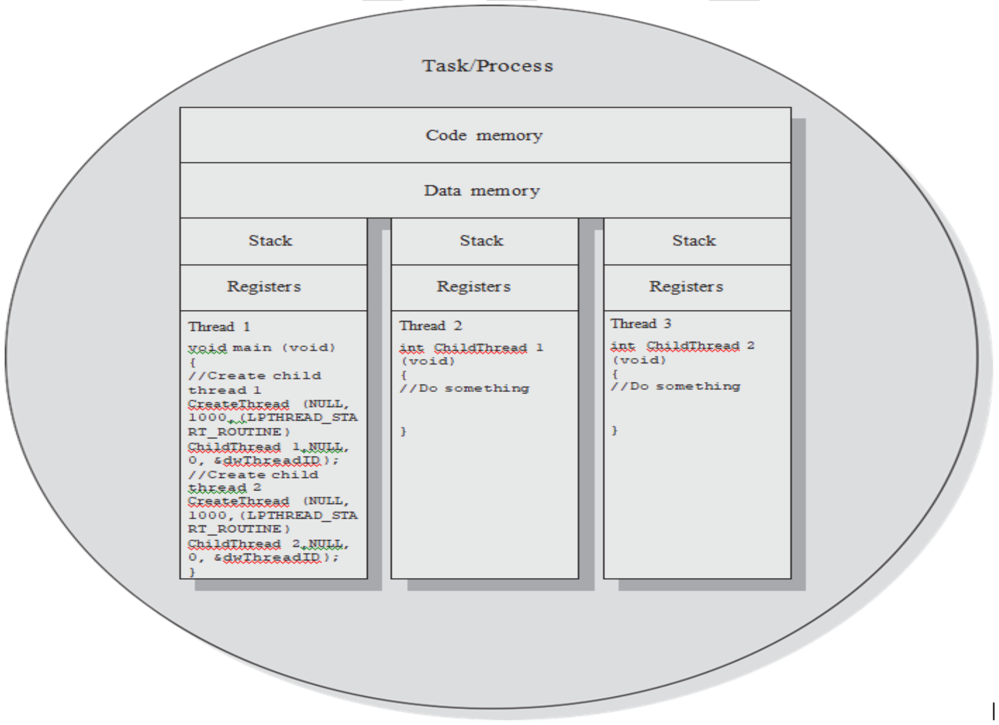
Fig. 10.6 Process states and state transition representation

- The state at which a process is being created is referred as 'Created State'. The Operating System recognises a process in the '*Created State*' but no resources are allocated to the process.
- The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as '*Ready State*'. At this stage, the process is placedin the '*Ready list*' queue maintained by the OS.

- The state where in the source code instructions corresponding to the process is being executed is called '*Running State*'. Running state is the stateat which the process execution happens.
- '*Blocked State/Wait State' refers to a state where a running process is temporarily suspended from execution* and does not have immediate access to resources. The blocked state might be invoked by various conditions like: the process enters a wait state for an event to occur (e.g. Waiting for user inputs such as keyboard input) or waiting for getting access to a shared resource.
- A statewhere the process completes its execution is known as '*Completed State*'.
- The transition of a process from one state to another is known as '*State transition*'.
- When a process changes its state from Ready to running or from running to blocked or terminated or from blocked to running, the CPU allocation for the process mayalso change.

5. Explain multithreading

- A process/task in embedded application may be a complex or lengthy one and it may contain various suboperations like getting input from I/O devices connected to the processor, performing some internal calculations/operations, updating some I/O devices etc. If all the sub functions of a task are executed in sequence, the CPU utilisation may not be efficient. For example, if the process is waiting for a user input, the CPU enters the wait state for the event, and the process execution also enters a wait state.
- Instead of this single sequential execution of the whole process, if the task/process is split into different threads carrying out the different subfunctionalities of the process, the CPU can be effectively utilised and when the thread corresponding to the I/O operation enters the wait state, another threads which do not require the I/O event for their operation can be switched into execution. This leads to more *speedy execution of the process* *and the efficient utilisation of the processor time and resources.*
- The multithreaded architecture of a process can be better visualised with the thread-process diagram shown in Fig. 10.8.



```
void main (void)
{
//Create child
thread 1
CreateThread (NULL,
1000,(LPTHREAD_STA
RT_ROUTINE)
ChildThread 1,NULL,
0, &dwThreadID);
//Create child
thread 2
CreateThread (NULL,
1000,(LPTHREAD_STA
RT_ROUTINE)
ChildThread 2,NULL,
0, &dwThreadID);
}
```

```
int ChildThread 1
(void)
{
//Do something


}
```

```
int ChildThread 2
(void)
{
//Do something


}
```

Fig. 10.8 Process with multi-threads

- If the process is split into multiple threads, which executes a portion of the process, there will be a main thread and rest of the threads will be created within the main thread.

Use of multiple threads to execute a process brings the following advantage.

➢ Better memory utilisation. Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.

- ➢ Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilised by other threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of the process.
- ➢ Efficient CPU utilisation. The CPU is engaged all time.

6. Differentiate between Multiprocessing and Multitasking

Multiprocessing:

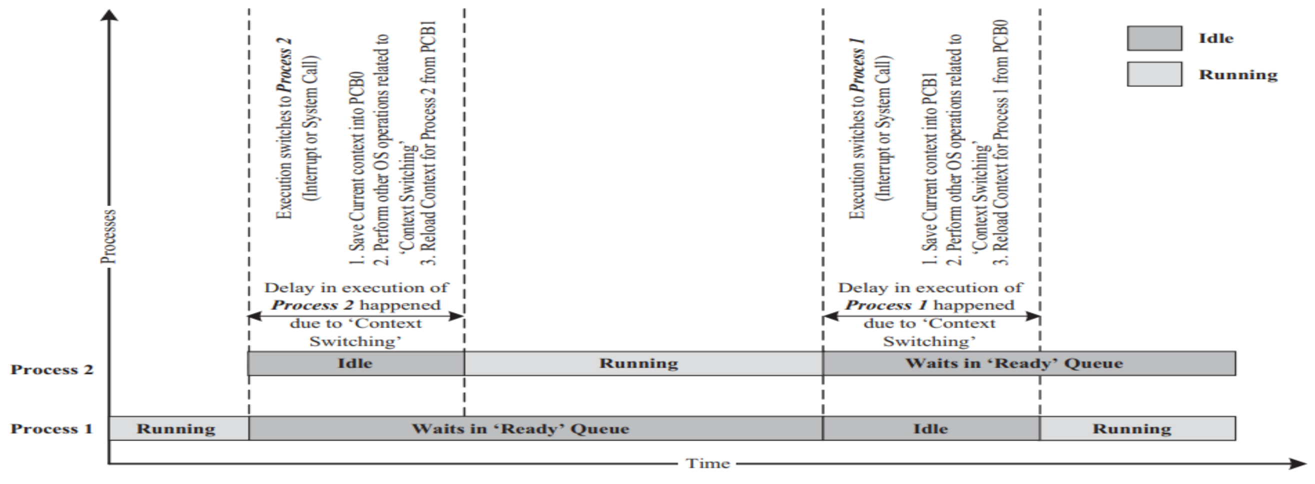
- Systems which are capable of performing ***multiprocessing***, are known as *multiprocessor* systems. *Multiprocessor* systems possess multiple CPUs and can execute multiple processes simultaneously.

Multitasking:

- The ability of the operating system to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*. In a uniprocessor system, it is not possible to execute multiple processes simultaneously. However, it is possible for a uniprocessor system to achieve some degree of pseudo parallelism in the execution of multiple processes by switching the execution amongdifferent processes.
- The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as ***multitasking*.**

Context Switching

- Multitasking creates the illusion of multiple tasks executing in parallel. Multitasking involves the switching of CPU fromexecuting one task to another.
- In a multitasking environment, when task/process switching happens, the virtual processor (task/process) gets its properties converted into that of the physical processor.
- The switching of the virtual processor to physical processor is controlled by the scheduler of the OS kernel. Whenever a CPU switching happens, the current context of execution should be saved to retrieve it at a later point of time when the CPU executes the process, which is interrupted currentlydue to execution switching.



Processes
Execution switches to Process 2 (Interrupt or System Call)
1. Save Current context into PCB0
2. Perform other OS operations related to 'Context Switching'
3. Reload Context for Process 2 from PCB1
Execution switches to Process 1 (Interrupt or System Call)
1. Save Current context into PCB1
2. Perform other OS operations related to 'Context Switching'
3. Reload Context for Process 1 from PCB0
Idle
Running
Delay in execution of Process 2 happened due to 'Context Switching'
Delay in execution of Process 1 happened due to 'Context Switching'
Process 2
Idle
Running
Waits in 'Ready' Queue
Process 1
Running
Waits in 'Ready' Queue
Idle
Running
Time

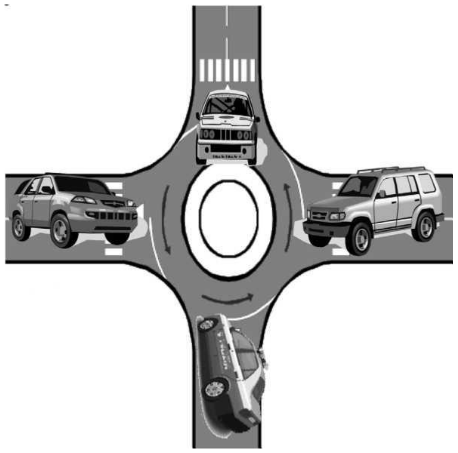
Fig. 10.11 Context switching

- The context saving and retrieval is essential for resuming a process exactly from the point where it was interrupted due to CPU switching.
  - ✓ The act of switching CPU among the processes or changing the current execution context is known as '*Context switching*'.
  - ✓ The act of saving the current context which contains the context details (Register details, memory details, system resource usage details, execution details, etc.) for the currently running process at the time of CPU switching is known as '*Context saving*'.
  - ✓ The process of retrieving the saved context details for a process, which is going to be executed due to CPU switching, is known as '*Context retrieval*'. Multitasking involves '*Context switching*' (Fig. 10.11), '*Context saving*' and '*Context retrieval*'.

7. Explain the concept of deadlock with a neat diagram and mention how to avoid deadlock

- In a multiprogramming environment several processes may compete for a finite number of resources. A process request resources; if the resource is available at that time a process enters the wait state. Waiting process may never change its state because the resources requested are held by other waiting process. This situation is known as ***deadlock.***

Deadlock Characteristics: In a deadlock process never finish executing and system resources are tied up. A deadlock situation can arise if the following four conditions hold simultaneously in a system.



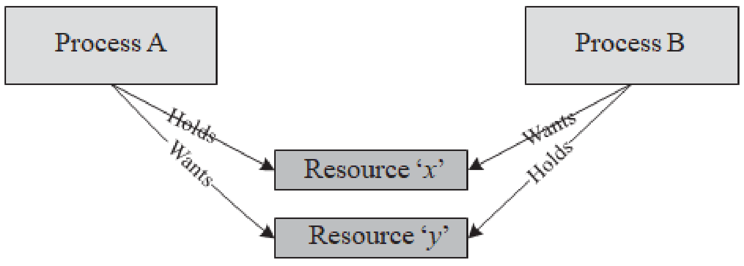


Fig. 10.25 Scenarios leading to deadlock

- ➢ **Mutual Exclusion**: At a time only one process can use the resources. If another process requests that resource, requesting process must wait until the resource has been released.
- ➢ **Hold and wait:** A process must be holding at least one resource and waiting to additional resource that is currently held by other processes.
- ➢ **No Preemption**: Resources allocated to a process can't be forcibly taken out from it, unless it releases that resource after completing the task.
- ➢ **Circular Wait**: A set {P0, P1, …….Pn} of waiting state/ process must exists such that P0 is waiting for a resource that is held by P1, P1 is waiting for the resource that is held by P2 ….. P(n – 1) is waiting for the resource that is held by Pn and Pn is waiting for the resources that is held by P4.

To avoid deadlock:

- ➢ Deadlock Handling: Smart OS
- ➢ Ignore Deadlocks: Deadlock free
- ➢ Detect & Recover: Traffic light signal mechanism
- ➢ Avoid deadlock: Careful resource allocation
- ➢ Prevent Deadlocks: Sleep & Wakeup

8. Write a note on Message passing

Message passing is an

- Synchronous information exchange mechanism used for Inter Process/Thread Communication.
- The major difference between shared memory and message passing technique is that, through shared memory lots of data can be shared whereas only limited amount of info/data is passed through message passing.
- Also message passing is relatively fast and free from the synchronisation overheads compared to shared memory.

Based on the message passing operation between the processes, message passing is classified into

- ***Message Queues:*** Process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called „*Message queue*", which stores the messages temporarily in a system defined memory object, to pass it to the desired process. Messages are sent and received through *send (Name of the process to which the message is to be sent, message*) and *receive* (*Name of the process from which the message is to be received, message*) methods.

The messages are exchanged through a message queue. The implementation of the message queue, send and receive methods are OS kernel dependent

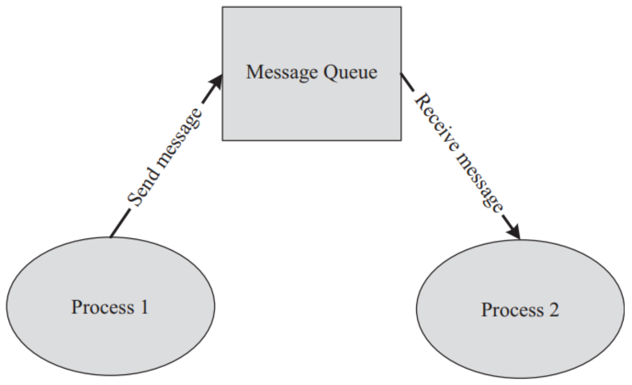


Fig. 10.20 Concept of message queue based indirect messaging for IPC

- ***Mailbox:*** Mailbox is a special implementation of message queue. Usually used for one way communication, only a single message is exchanged through mailbox whereas „message queue" can be used for exchanging multiple messages. One task/process creates the mailbox and other tasks/process can subscribe to this mailbox for getting message notification. The implementation of the mailbox is OS kernel dependent. The MicroC/ OS-II RTOS implements mailbox as a mechanism for inter task communication

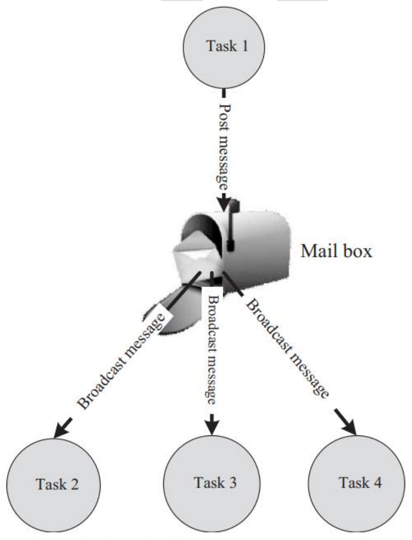


Fig. 10.21 Concept of Mailbox based indirect messaging for IPC

- ***Signalling:*** Signals are used for an asynchronous notification mechanism. The signal mainly used for the execution synchronization of tasks process/ tasks. Signals do not carry any data and are not queued. The implementation of signals is OS kernel dependent and VxWorks RTOS kernel implements „signals" for inter process communication.

9. Explain the concept of Semaphore

- Semaphore is a sleep and wakeup based mutual exclusion implementation for shared resourceaccess.
- Semaphore is a system resource and the process which wants to access the shared resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently acquired by it.
- The resources which are shared among a process can be either for exclusive use by a process or for using by a number of processes at a time.
  - ➢ The display device of an embedded system is a typical example for the shared resource which needs exclusive access by a process.
  - ➢ The Hard disk (secondary storage) of a system is a typical example for sharing the resource among a limited number of multiple processes. Various processes can access the different sectors of the hard-disk concurrently.

Based on the implementation of the sharing limitation of the shared resource, semaphores are classified into two; namely '*Counting Semaphore*' and '*Binary Semaphore*'.

- **The '*CountingSemaphore*'**
  - It limits the access of resources by a fixed number of processes/threads.
  - It maintains a count between zero and a maximum value.
  - It limits the usage of the resource to the maximum value of the count supported by it.
  - A real world example for the counting semaphore concept is the *dormitory system* for accommodation (Fig. 10.34). A dormitory contains a fixed number of beds (say 5) and at any point of time it can be shared by the maximum number of users supported by the dormitory. If a person wants to avail the dormitory facility, he/she can contact the dormitory caretaker for checking the availability. If beds are available in the dorm the caretaker will hand over the keys to the user. If beds are not available currently, the user can register his/her name to get notifications when a slot is available. Those who are availing the dormitory shares the dorm facilities like TV, telephone, toilet, etc. When a dorm user vacates, he/she gives the keys back to the caretaker. The caretaker informs the users, who booked in advance, about the dorm availability.

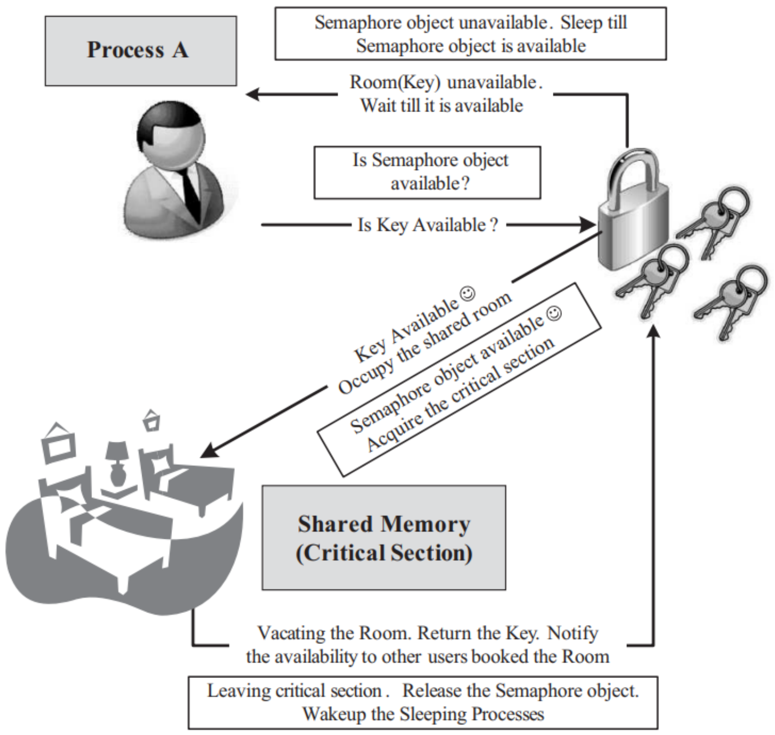

Process A
Semaphore object unavailable. Sleep till Semaphore object is available
Room(Key) unavailable. Wait till it is available
Is Semaphore object available?
Is Key Available ?
Key Available ☺ Occupy the shared room
Semaphore object available ☺ Acquire the critical section
Shared Memory (Critical Section)
Vacating the Room. Return the Key. Notify the availability to other users booked the Room
Leaving critical section. Release the Semaphore object. Wakeup the Sleeping Processes

Fig. 10.34 The Concept of Counting Semaphore

- **The 'binary semaphore'**
  - Provides exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it when it is being owned by a process.
  - The implementation of binary semaphore is OS kernel dependent. Under certain OS kernel it is referred as *mutex*.
  - Any process/thread can create a '*mutex object*' and other processes/threads of the system can use this '*mutex object*' for synchronising the access to critical sections.
  - Only one process/thread can own the '*mutex object*' at a time.
  - A real world example for the mutex concept is the hotel accommodation system (*lodging system*) Fig. 10.35. The rooms in a hotel are shared for the public. Any user who pays and follows the norms of the hotel can avail the rooms for accommodation. A person wants to avail the hotel room facility can contact the hotel reception for checking the room availability (see Fig. 10.35). If room is available the receptionist will handover the room key to the user. If room is not available currently, the user can book the room to get notifications when a room is available. When a person gets a room he/she is granted the exclusive access to the room facilities like TV, telephone, toilet, etc. When a user vacates the room, he/she gives the keys back to the receptionist. The receptionist informs the users, who booked in advance, about the room's availability

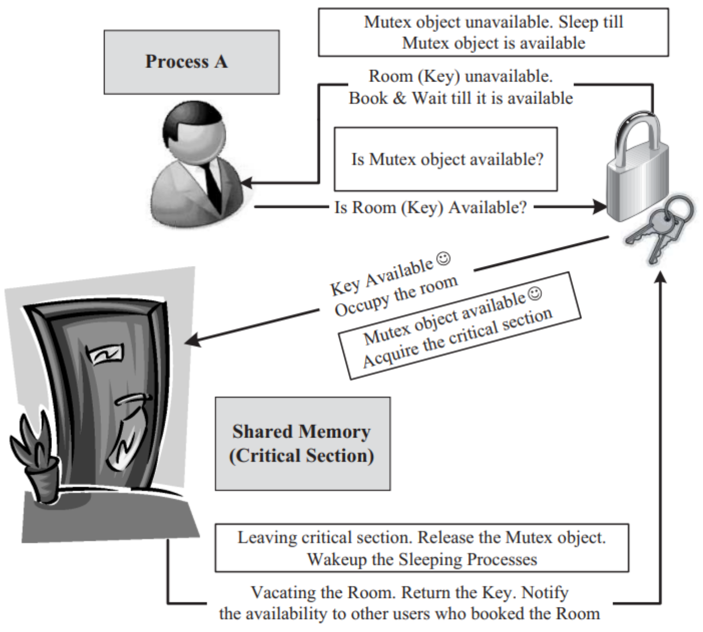

Process A
Mutex object unavailable. Sleep till Mutex object is available
Room (Key) unavailable. Book & Wait till it is available
Is Mutex object available?
Is Room (Key) Available?
Key Available ☺ Occupy the room
Mutex object available ☺ Acquire the critical section
Shared Memory (Critical Section)
Leaving critical section. Release the Mutex object. Wakeup the Sleeping Processes
Vacating the Room. Return the Key. Notify the availability to other users who booked the Room

Fig. 10.35 The Concept of Binary Semaphore (Mutex)

10. Explain the functional and non functional requirements for selecting RTOS for an embedded system

Functional Requirements

- **Processor Support** It is not necessary that all RTOS's support all kinds of processor architecture. It is essential to ensure the processor support by the RTOS.
- **Memory Requirements** The OS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like FLASH. OS also requires working memory RAM for loading the OS services. Since embedded systems are memory constrained, it is essential to evaluate the minimal ROM and RAM requirements for the OS under consideration.
- **Real-time Capabilities** It is not mandatory that the operating system for all embedded systems need to be Real-time and all embedded Operating systems are 'Real-time' in behaviour. The task/process scheduling policies plays an important role in the 'Real-time' behaviour of an OS. Analyse the real-time capabilities of the OS under consideration and the standards met by the operating system for real-time capabilities.
- **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.
- **Inter Process Communication and Task Synchronisation** The implementation of Inter Process Communication and Synchronisation is OS kernel dependent. Certain kernels may provide a bunch of options whereas others provide very limited options. Certain kernels implement policies for avoiding priorityinversion issues in resource sharing.
- **Modularisation** Support Most of the operating systems provide a bunch of features. At times it may not be necessary for an embedded product for its functioning. It is very useful if the OS supports modularisation where in which the developer can choose the essential modules and re-compile the OS image for functioning. Windows CE is an example for a highly modular operating system.
- **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.
- **Development Language Support** Certain operating systems include the run time libraries required for running applications written in languages like Java and C#. A Java Virtual Machine (JVM) customised for the Operating System is essential for running java applications. Similarly the .NET Compact Framework (.NETCF) is required for running Microsoft® .NET applications on top of the Operating System. The OS may include these components as built-in component, if not, check the availability of the same from a third party vendor for the OS under consideration.

Non-functional Requirements

- **Custom Developed or Off the Shelf** Depending on the OS requirement, it is possible to go for the complete development of an operating system suiting the embedded system needs or use an off the shelf, readily available operating system, which is either a commercial product or an Open Source product, which is in close match with the system requirements. Sometimes it may be possible to build the required features by customising 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.
- **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.
- **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 Operating Systems may be superior in performance, but the availability of tools for supporting the development may be limited. Explore the different tools available for the OS under consideration.
- **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.
- **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 analysed thoroughly.

11. Explain the role of Integrated Development Environment (IDE) for Embedded Software Development

- In embedded system development context, Integrated Development Environment (IDE) stands for an integrated environment for developing and debugging the target processor specific embedded firmware.
- IDE is a software package which bundles a ‘Text Editor (Source Code Editor)’, ‘Cross-compiler (for cross platform development and compiler for same
- IDEs used in embedded firmware development are slightly different from the generic IDEs used for high level language based development for desktop applications.

In Embedded Applications, the IDE is either supplied by the target processor/controller manufacturer or by third party vendors or as Open Source.

- ✓ MPLAB is an IDE tool supplied by microchip for developing embedded firmware using their PIC family of microcontrollers.
- ✓ Keil µVision5 from ARMKeil is an example for a third party IDE, which is used for developing embedded firmware for *ARM* family microcontrollers.
- ✓ CodeWarriorDevelopment Studio is an IDE for ARM family of processors/MCUs and DSP chips from Freescale.

- It should be noted that in embedded firmware development applications each IDE is designed for a specific family of controllers/processors and it may not be possible to develop firmware for all family of controllers/processors using a single IDE
- However there is a rapid move happening towards the open source IDE, Eclipse for embedded development. Most of the proccessor/control manufacturers and third party IDE providers are trying to build the IDE around the popular Eclipse open source IDE.

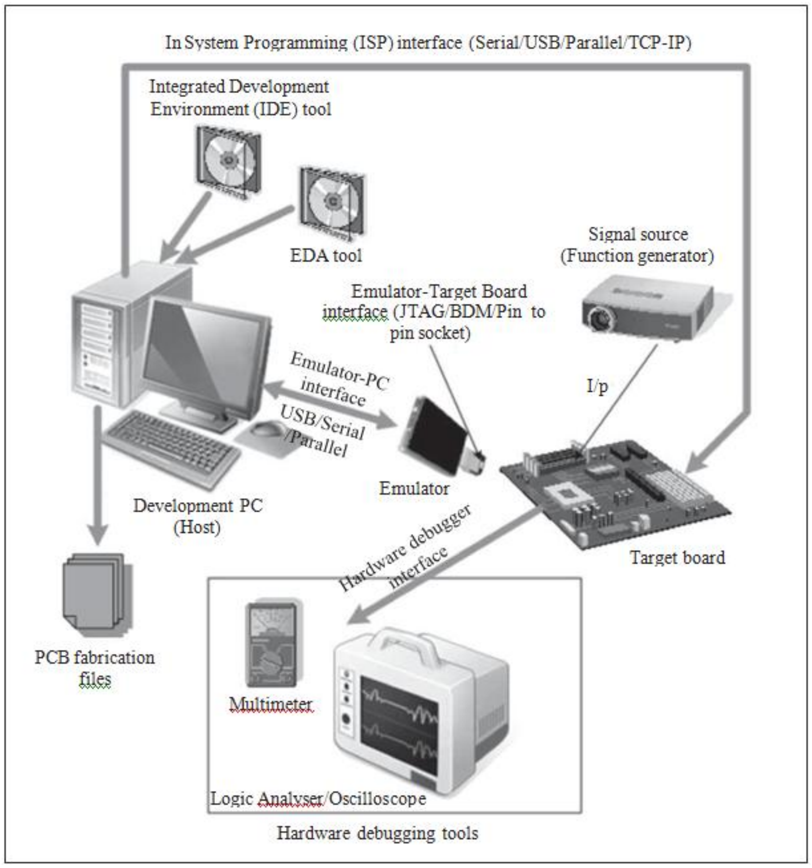

In System Programming (ISP) interface (Serial/USB/Parallel/TCP-IP)
Integrated Development Environment (IDE) tool
EDA tool
Signal source (Function generator)
Emulator-Target Board interface (JTAG/BDM/Pin to pin socket)
Emulator-PC interface USB/Serial /Parallel
I/p
Emulator
Development PC (Host)
Target board
Hardware debugger interface
PCB fabrication files
Multimeter
Logic Analyser/Oscilloscope
Hardware debugging tools

Fig. 13.1 The Embedded System Development Environment

12. Explain boundary scanning for hardware testing with diagram

- As the complexity of the hardware increase, the number of chips present in the board and the interconnection among them may also increase.
- The device packages used in the PCB become miniature to reduce the total board space occupied by them and multiple layers may be required to route the interconnections among the chips. With miniature device packages and multiple layers for the PCB it will be very difficult to debug the hardware using magnifying glass, multimeter, etc. to check the interconnection among the various chips.
- Boundary scan is a technique used for testing the interconnection among the various chips, which support JTAG interface, present in the board.

Chips which support boundaryscan associate a boundary scan cell with each pin of the device.

- ✓ A JTAG port which contains the five signal lines namely TDI, TDO, TCK, TRST and TMS form the Test Access Port (TAP) for a JTAG supported chip.Each device will have its own TAP.
- ✓ The PCB also contains a TAP for connecting the JTAG signal lines to the external world. A boundary scan path is formed inside the board by interconnecting the devices throughJTAG signal lines.
- ✓ The TDI pin of the TAP of the PCB is connected to the TDI pin of the first device.
- ✓ The TDO pin of the first device is connected to the TDI pin of the second device. In this way all devices are interconnected and the TDO pin of the last JTAG device is connected to the TDO pin of the TAP of the PCB.
- ✓ The clock line TCK and the Test Mode Select (TMS) line of the devices are connected to the clock line and Test mode select line of the Test Access Port of the PCB respectively. This forms a boundary scan path. Figure 13.41 illustrates the same.

- As mentioned earlier, each pin of the device associates a boundary scan cell with it. The boundary scan cell is a multipurpose memory cell. The boundary scan cell associated with the input pins of an IC is known as 'input cells' and the boundary scan cells associated with the output pins of an IC is known as 'output cells'.

- The boundary scan cells can be used for capturing the input pin signal state and passing it to the internal circuitry, capturing the signals from the internal circuitry and passing it to the output pin, and shifting the data received from the Test Data In pin of the TAP.
- The boundary scan cells associated with the pins are interconnected and they form a chain from the TDI pin of the device to its TDO pin.

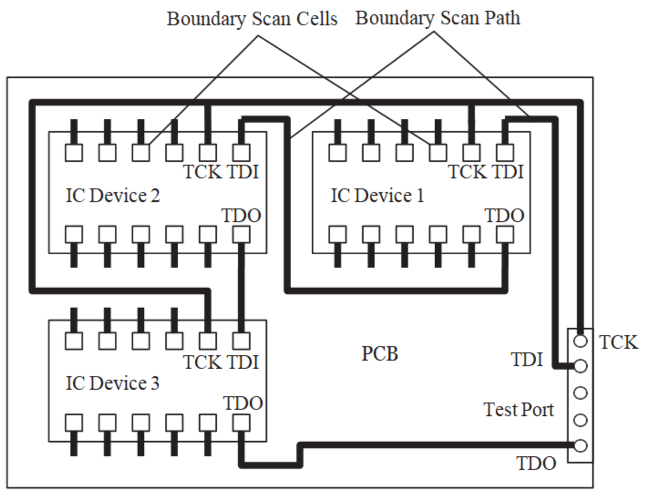


Fig. 13.41 JTAG based boundary scanning for hardware testing

The boundary scan cells can be operated in Normal, Capture, Update and Shift modes.

- **In the Normal mode**, the input of the boundaryscan cell appears directly at its output.
- **In the Capture mode**, the boundary scan cell associated with each input pin of the chip captures the signal from the respective pins to the cell and the boundary scan cell associated with each output pin of the chip captures the signal from the internal circuitry. In the
- **Update mode**, the boundary scan cell associated with each input pin of the chip passes the already captured data to the internal circuitry and the boundary scan cell associated with each output pin of the chip passes the already captured data to the respective output pin.
- **In the shift mode**, data is shifted from TDI pin to TDO pin of the device through the boundary scan cells. ICs supporting boundary scan contain additional boundary scan related registers for facilitating the boundary scan operation. Instruction Register, Bypass Register, Identification Register, etc. are examples of boundary scan related registers.

13. Write a note on a. Disassembler b. Decompiler c. Debugging d. Emulator e. Simulator

- **Disassembler** is a utility program which converts machine codes into target processor specific Assembly codes/instructions. The process of convertingmachine codes into Assembly code is known as ‘Disassembling’.

- **De-compilers** reproduce the code in a high level language. Frequently, this high level language is C, because C is simple and primitive enough to facilitate the decompilation process. Decompilation does have its drawbacks, because lots of data and readability constructs are lost during the original compilation process, and they cannot be reproduced. Since the science of decompilation is still young, and results are "good" but not "great”.

- **Debugging** in embedded application is the process of diagnosing the firmware execution, monitoring the target processor’s registers and memory while the firmware is running and checking the signals from various buses of the embedded hardware.
  - ✓ Debugging process in embedded application is broadly classified into two, namely; hardware debugging and firmware debugging.
  - ✓ Hardware debugging deals with the monitoring of various bus signals and checking the status lines of the target hardware.
  - ✓ Firmware debugging deals with examining the firmware execution, execution flow, changes to various CPU registers and status registers on execution of the firmware to ensure that the firmware is runningas per the design.

- **Emulator** is a self-contained hardware device which emulates the target CPU. The emulator hardware contains necessary emulation logic and it is hooked to the debugging application running on the development PC on one end and connects to the target board through some interface on the other end.

- **Simulator** is a software application that precisely duplicates (mimics) the target CPU and simulates the various features and instructions supported by the target CPU.