MODULE – 5

RTOS and IDE for Embedded System Design

Operating System Basics

Operating System Basics

- 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.
- The primary functions of an operating system are:
  - Make the system convenient to use
  - Organise and manage the system resources efficiently and correctly

Operating System Architecture

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

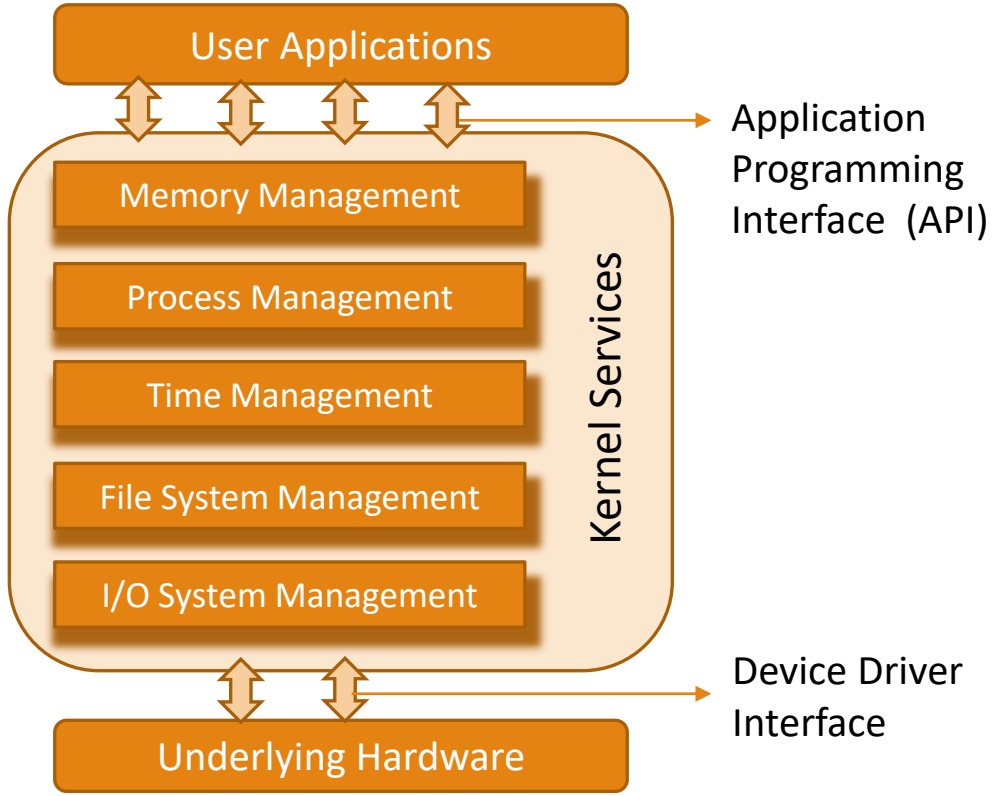


Fig: The Operating System Architecture

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 between system resources and user applications.
- Kernel contains a set of system libraries and services.

The Kernel (continued)

- For a general purpose OS, the kernel contains different services for handling the following:
  - Process Management
  - Primary Memory Management
  - File System Management
  - I/O System (Device) Management
  - Secondary Storage Management
  - Protection Systems
  - Interrupt Handler

Process Management

- Process management deals with managing the processes/tasks.
- Process management 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.
- 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
- The various file system management operations are OS dependent.
  - For example, the kernel of Microsoft DOS OS supports a specific set of file system management operations and they are not the same as the file system operations supported by UNIX Kernel.

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 Programming Interfaces (APIs) exposed by the kernel.
- The kernel maintains a list of all the I/O devices of the system.
  - May be available in advance or updated dynamically as and when a new device is installed.
- 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.
- 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 in disks (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. ‘Administrator’, ‘Standard’, ‘Restricted’ permissions in Windows XP.
- 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 or portions of another user's data or profile details.
- In addition, some application may not 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.

Kernel Space and User Space

- The applications/services are classified into two categories:
  - User applications
  - Kernel applications
- Kernel Space is the memory space at which the kernel code is located
  - Kernel applications/services are kept in this contiguous area of primary (working) memory.
  - It is protected from the unauthorised access by user programs/applications.
- User Space is the memory area where user applications are loaded and executed.

Kernel Space and User Space (continued)

- The partitioning of memory into kernel and user space is purely OS dependent.
  - Some OS implement this kind of partitioning and protection whereas some OS do not segregate the kernel and user application code storage into two separate areas.
- In an operating system with virtual memory support, the user applications are loaded into its corresponding virtual memory space with *demand paging technique*.
  - The entire code for the user application need not be loaded to the main (primary) memory at once.
  - The user application code is split into different pages and these pages are loaded into and out of the main memory area on a need basis.
  - The act of loading the code into and out of the main memory is termed as '*Swapping*'.
  - Swapping happens between the main (primary) memory and secondary storage memory.

Monolithic Kernel and Microkernel

- The kernel forms the heart of an operating system.
- Different approaches are adopted for building an Operating System kernel.
- Based on the kernel design, kernels can be classified into
  - Monolithic Kernel
  - Microkernel

Monolithic Kernel

- In monolithic kernel architecture, all kernel services run in the kernel space.
- Here all kernel modules run within the same memory space under a single kernel thread.
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilisation of the low-level features of the underlying system.
- The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application.
- LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel.

Monolithic Kernel (continued)

- The architecture representation of a monolithic kernel is given in the figure.

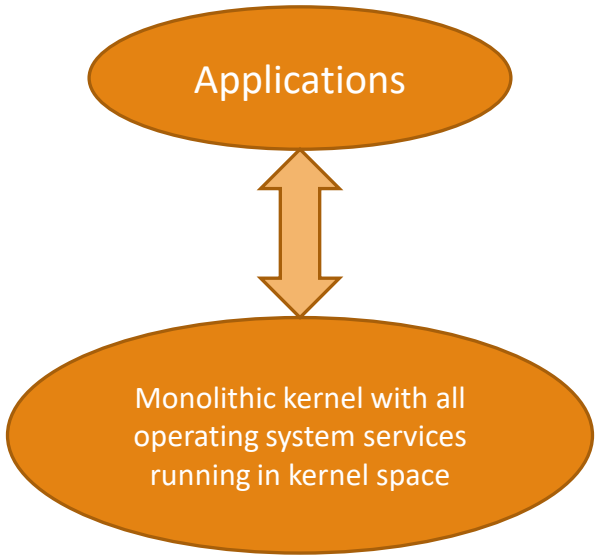


Fig: The Monolithic Kernel Model

Microkernel

- The microkernel design incorporates only the essential set of Operating System services into the kernel.
- The rest of the Operating System services are implemented in programs known as '*Servers*' which runs in user space.
- This provides a 'highly modular design and OS-neutral abstract to the kernel.
- Memory management, process management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel.
- Mach, QNX, Minix 3 kernels are examples for microkernel.

Microkernel (continued)

- The architecture representation of a microkernel is shown in the figure.

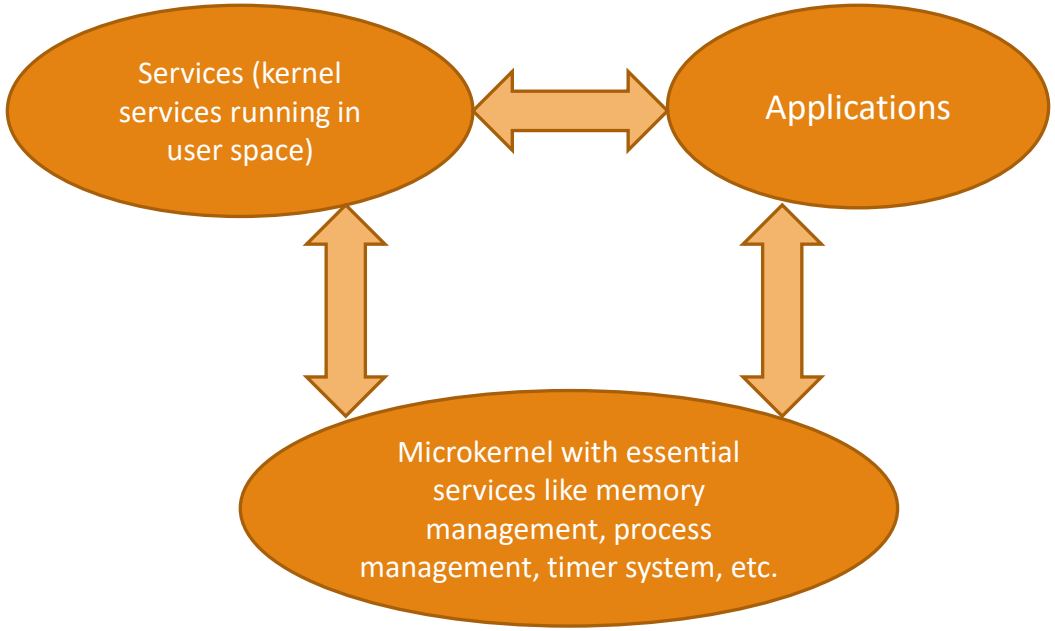


Fig: The Microkernel Model

Microkernel (continued)

- Microkernel based design approach offers the following benefits:
  - Robustness
    - If a problem is encountered in any of the services, which runs as *'Server'* application, the same can be reconfigured and re-started without the need for re-starting the entire OS.
    - Thus, this approach is highly useful for systems, which demands high *'availability'*.
    - Since the services which run as *'Servers'* are running on a different memory space, the chances of corruption of kernel services are ideally zero.
  - Configurability
    - Any service which runs as *'Server'* application can be changed without the need to restart the whole system.
    - This makes the system dynamically configurable.

Types of Operating Systems

Types of Operating Systems

- Depending on the type of kernel and kernel services, purpose and type of computing systems where the OS is deployed and the responsiveness to applications, Operating Systems are classified into different types.
  - General Purpose Operating System (GPOS)
  - Real-Time Operating System (RTOS)

General Purpose Operating System (GPOS)

- The operating systems which are deployed in general computing systems are referred as *General Purpose Operating Systems (GPOS)*.
- The kernel of such a GPOS is more generalised and it contains all kinds of services required for executing generic applications.
- General purpose operating systems are often quite non-deterministic in behaviour.
  - Their services can inject random delays into application software and may cause slow responsiveness of an application at unexpected times.
- GPOS are usually deployed in computing systems where deterministic behaviour is not an important criterion.
- Personal Computer/Desktop system is a typical example for a system where GPOSs are deployed.
- Windows XP/MS-DOS etc. are examples for General Purpose Operating Systems.

Real-Time Operating System (RTOS)

- '*Real-Time*' implies deterministic timing behaviour.
  - Deterministic timing behaviour in RTOS context means the OS services consumes only known and expected amounts of time regardless the number of services.
- A Real-Time Operating System or RTOS implements policies and rules concerning time-critical allocation of a system's resources.
  - The RTOS decides which applications should run in which order and how much time needs to be allocated for each application.
- Predictable performance is the hallmark of a well-designed RTOS.
- This is best achieved by the consistent application of policies and rules.
  - Policies guide the design of an RTOS.
  - Rules implement those policies and resolve policy conflicts.
- Windows CE, QNX, VxWorks, MicroC/OS-II, etc. are examples of Real-Time Operating Systems (RTOS).

The Real-Time Kernel

- The kernel of a Real-Time Operating System is referred as *Real-Time kernel*.
- The Real-Time kernel is highly specialised and it contains only the minimal set of services required for running the user applications/tasks.
- The basic functions of a Real-Time kernel are:
  - Task/Process Management
  - Task/Process Scheduling
  - Task/Process Synchronisation
  - Error/Exception Handling
  - Memory Management
  - Interrupt Handling
  - Time Management

The Real-Time Kernel (continued)

- Task/Process Management
  - Deals with
    - setting up the memory space for the tasks
    - loading the task's code into the memory space
    - allocating system resources
    - setting up a Task Control Block (TCB) for the task
    - task/process termination/deletion
  - A *Task Control Block (TCB)* is used for holding the information corresponding to a task.

The Real-Time Kernel (continued)

- TCB usually contains the following set of information:
  - ***Task ID***: Task Identification Number
  - ***Task State***: The current state of the task (e.g. State = 'Ready' for a task which is ready to execute)
  - ***Task Type***: Indicates what is the type for this task. The task can be a hard real time or soft real time or background task.
  - ***Task Priority***: Task priority (e.g. Task priority = 1 for task with priority = 1)
  - ***Task Context Pointer***: Pointer for context saving
  - ***Task Memory Pointers***: Pointers to the code memory, data memory and stack memory for the task
  - ***Task System Resource Pointers***: Pointers to system resources (semaphores, mutex, etc.) used by the task
  - ***Task Pointers***: Pointers to other TCBs (TCBs for preceding, next and waiting tasks)
  - ***Other Parameters***: Other relevant task parameters

The Real-Time Kernel (continued)

- The parameters and implementation of the TCB is kernel dependent.
- The TCB parameters vary across different kernels, based on the task management implementation.
- Task management service utilises the TCB of a task in the following way:
  - Creates a TCB for a task on creating a task
  - Delete/remove the TCB of a task when the task is terminated or deleted
  - Reads the TCB to get the state of a task
  - Update the TCB with updated parameters on need basis (e.g. on a context switch)
  - Modify the TCB to change the priority of the task dynamically

The Real-Time Kernel (continued)

- Task/Process Scheduling
  - Deals with sharing the CPU among various tasks/processes.
  - A kernel application called *'Scheduler'* handles the task scheduling.
  - *Scheduler* is nothing but an algorithm implementation, which performs the efficient and optimum scheduling of tasks to provide a deterministic behaviour.
- Task/Process Synchronisation
  - Deals with synchronising the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.

The Real-Time Kernel (continued)

- Error/Exception Handling
  - Deals with registering and handling the errors occurred/exceptions raised during the execution of tasks.
  - Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution, etc. are examples of errors/exceptions.
  - Errors/Exceptions can happen at the kernel level services or at task level.
    - Deadlock is an example for kernel level exception, whereas timeout is an example for a task level exception.
    - The OS kernel gives the information about the error in the form of a system call (API).
    - Watchdog timer is a mechanism for handling the timeouts for tasks.

The Real-Time Kernel (continued)

- Memory Management
  - RTOS makes use of '*block*' based memory allocation technique, instead of the usual dynamic memory allocation techniques used by the GPOS.
  - RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis.
  - The blocks are stored in a *'Free Buffer Queue'*.
  - To achieve predictable timing and avoid the timing overheads, most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection.
    - RTOS kernels assume that the whole design is proven correct and protection is unnecessary.
    - Some commercial RTOS kernels allow memory protection as optional.

The Real-Time Kernel (continued)

- A few RTOS kernels implement *Virtual Memory* concept for memory allocation if the system supports secondary memory storage (like HDD and FLASH memory).
- In the *'block'* based memory allocation, a block of fixed memory is always allocated for tasks on need basis and it is taken as a unit.
  - Hence, there will not be any memory fragmentation issues.
- The *'block'* based memory allocation achieves deterministic behaviour with the trade of limited choice of memory chunk size and suboptimal memory usage.

The Real-Time Kernel (continued)

- Interrupt Handling
  - Deals with the handling of various types of interrupts.
  - Interrupts provide Real-Time behaviour to systems.
  - Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
  - Interrupts can be either *Synchronous* or *Asynchronous*.
  - **Synchronous interrupts:**
    - Occur in sync with the currently executing task.
    - Usually the software interrupts fall under this category.
    - Divide by zero, memory segmentation error, etc. are examples of synchronous interrupts.
    - For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task.

The Real-Time Kernel (continued)

- **Asynchronous interrupts:**
  - Occur at any point of execution of any task, and are not in sync with the currently executing task.
  - The interrupts generated by external devices (by asserting the interrupt line of the processor/controller to which the interrupt line of the device is connected) connected to the processor/controller, timer overflow interrupts, serial data reception/ transmission interrupts, etc. are examples for asynchronous interrupts.
  - For asynchronous interrupts, the interrupt handler is usually written as separate task and it runs in a different context.
  - Hence, a context switch happens while handling the asynchronous interrupts.
- Priority levels can be assigned to the interrupts and each interrupt can be enabled or disabled individually.
- Most of the RTOS kernel implements 'Nested Interrupts' architecture.
  - Interrupt nesting allows the pre-emption (interruption) of an Interrupt Service Routine (ISR), servicing an interrupt, by a high priority interrupt.

The Real-Time Kernel (continued)

- Time Management
  - Accurate time management is essential for providing precise time reference for all applications.
  - The time reference to kernel is provided by a high-resolution Real-Time Clock (RTC) hardware chip (hardware timer).
  - The hardware timer is programmed to interrupt the processor/controller at a fixed rate.
    - This timer interrupt is referred as *'Timer tick'* and is taken as the timing reference by the kernel.
  - The *'Timer tick'* interval may vary depending on the hardware timer.
    - Usually the *'Timer tick'* varies in the microseconds range.
  - The time parameters for tasks are expressed as the multiples of the *'Timer tick'.*

The Real-Time Kernel (continued)

- The System time is updated based on the *'Timer tick'*.
- If the System time register is 32 bits wide and the *'Timer tick'* interval is 1 microsecond, the System time register will reset in

$$2^{32} \times 10^{-6}\ seconds = \frac{2^{32} \times 10^{-6}}{24 \times 60 \times 60}\ Days = \sim 0.0497\ Days = 1.19\ Hours$$

- If the 'Timer tick' interval is 1 millisecond, the system time register will reset in

$$2^{32} \times 10^{-3}\ seconds = \frac{2^{32} \times 10^{-3}}{24 \times 60 \times 60}\ Days = 49.7\ Days = \sim 50\ Days$$

The Real-Time Kernel (continued)

- The *'Timer tick'* interrupt is handled by the 'Timer Interrupt' handler of kernel.
- The *'Timer tick'* interrupt can be utilised for implementing the following actions:
  - Save the current context (Context of the currently executing task).
  - Increment the System time register by one. Generate timing error and reset the System time register if the timer tick count is greater than the maximum range available for System time register.
  - Update the timers implemented in kernel (Increment or decrement the timer registers for each timer depending on the count direction setting for each register. Increment registers with count direction setting = 'count up' and decrement registers with count direction setting = 'count down').
  - Activate the periodic tasks, which are in the idle state.
  - Invoke the scheduler and schedule the tasks again based on the scheduling algorithm.
  - Delete all the terminated tasks and their associated data structures (TCBs).
  - Load the context for the first task in the ready queue. Due to the re-scheduling, the ready task might be changed to a new one from the task, which was preempted by the 'Timer Interrupt' task.

Hard Real-Time

- Real-Time Operating Systems that strictly adhere to the timing constraints for a task are referred as *'Hard Real-Time'* systems.
  - They 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 loss and irrecoverable damages to the system/users.
- Hard Real-Time systems emphasise the principle '*A late answer is a wrong answer*'.
- Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples for Hard Real-Time Systems.
  - Any delay in the deployment of the air bags makes the life of the passengers under threat.

Hard Real-Time (continued)

- Hard Real-Time Systems does not implement the virtual memory model for handling the memory.
  - This eliminates the delay in swapping in and out the code corresponding to the task to and from the primary memory.
- Most of the Hard Real-Time Systems are automatic and does not contain a *Human in the Loop (HITL)*.
  - The presence of *human in the loop* for tasks introduces unexpected delays in the task execution.

Soft Real-Time

- Real-Time Operating Systems that do 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).
- A Soft Real-Time system emphasises the principle *'A late answer is an acceptable answer, but it could have done bit faster'*.
- Soft Real-Time systems most often have a *human in the loop (HITL)*.
- Automatic Teller Machine (ATM) is a typical example for Soft-Real-Time System.
  - If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.
- An audio-video playback system is another example for Soft Real-Time system.
  - No potential damage arises if a sample comes late by fraction of a second, for playback.

Tasks, Process and Threads

Task

- The term *'task'* refers to something that needs to be done.
- In the operating system context, a *task* 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' in the operating system context.
- 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 context and most often they are used interchangeably.

Process

- 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.

The Structure of a Process

- 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 visualised as shown in the figure.

The Structure of a Process (continued)

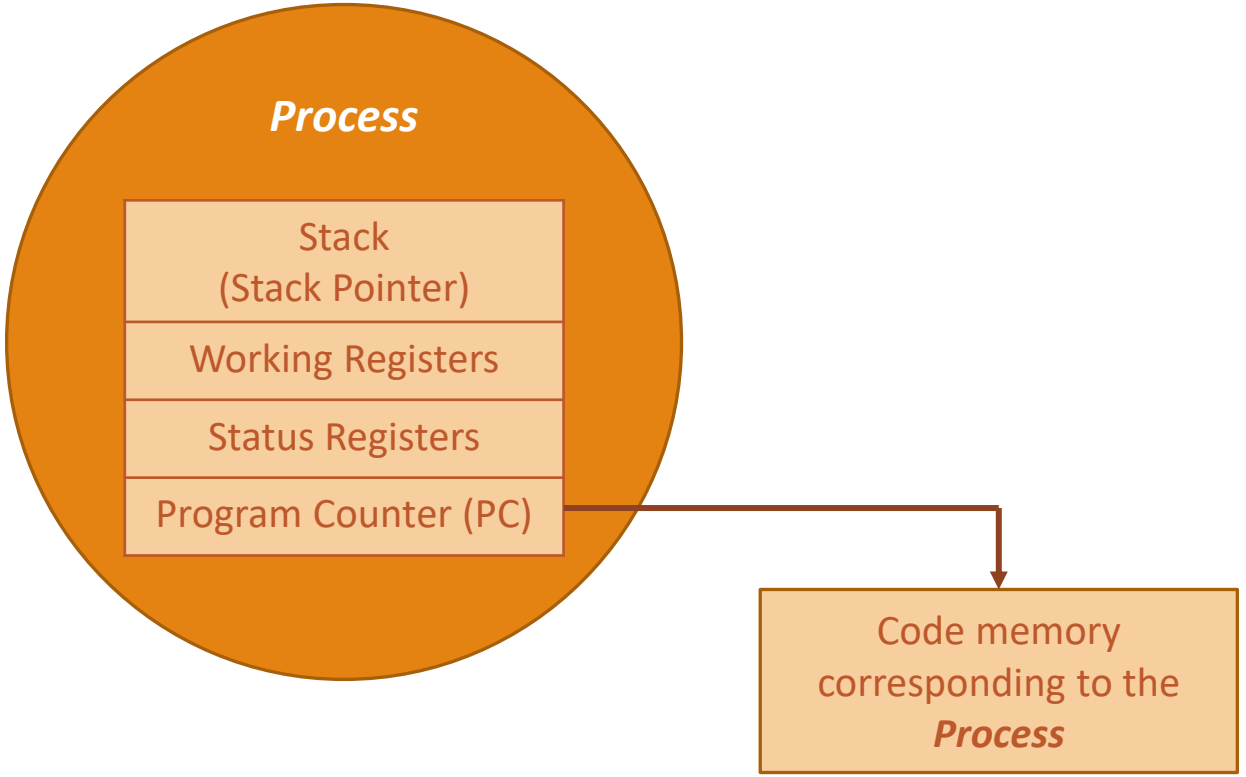


Fig: Structure of a Process

The Structure of a Process (continued)

- 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.

The Structure of a Process (continued)

- From a memory perspective, the memory occupied by the process is segregated into three regions as shown in the figure:
  - Stack memory - holds all temporary data such as variables local to the process
  - Data memory - holds all global data for the process
  - Code memory - contains the program code (instructions) corresponding to the process

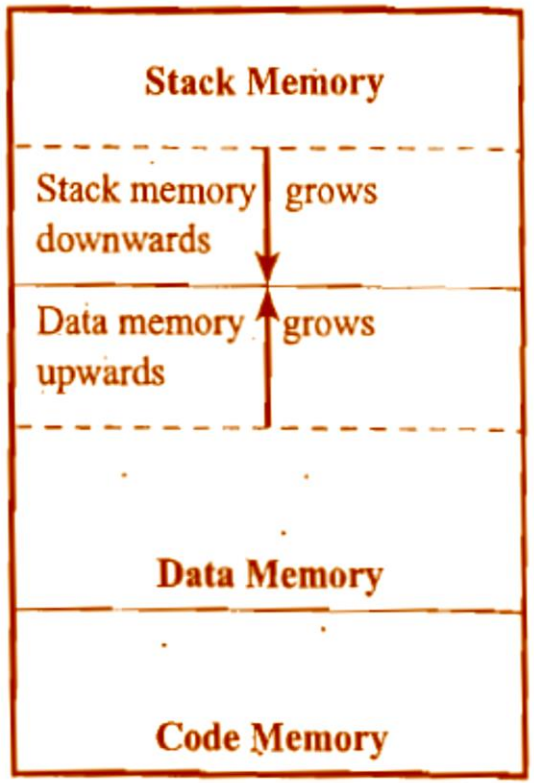


Fig: Memory Organisation of a Process

Process States and State Transition

- 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.
- The transition of a process from one state to another is known as *'State transition'*.
- Figure represents the various states and state transitions associated with a process.

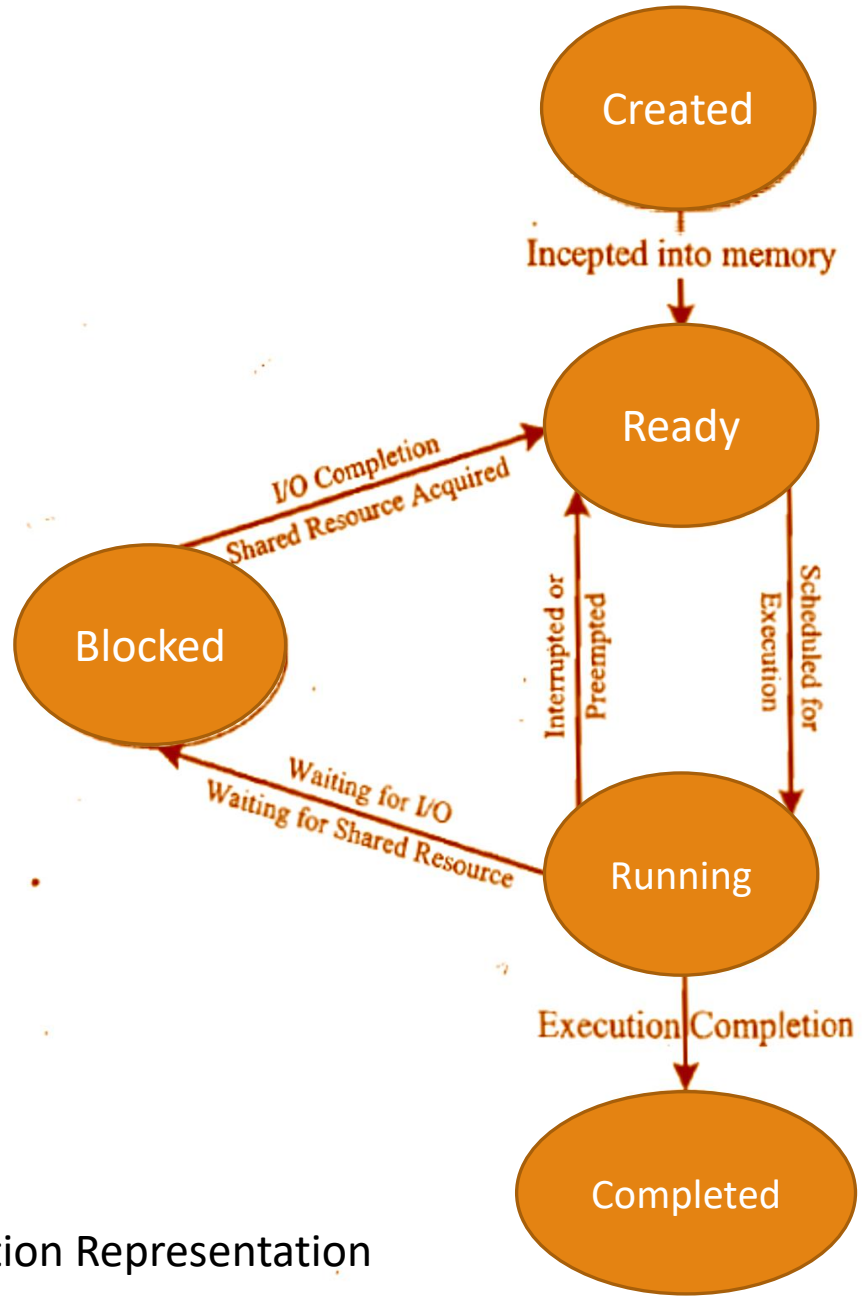


Fig: Process States and State Transition Representation

Process States and State Transition (continued)

- 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 placed in 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 state at which the process execution happens.

Process States and State Transition (continued)

- '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 state where the process completes its execution is known as 'Completed State’.

Process Management

- Process management deals with
  - creation of a process
  - setting up the memory space for the process
  - loading the process's code into the memory space
  - allocating system resources
  - setting up a Process Control Block (PCB) for the process
  - process termination/deletion

Threads

- 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 light-weight 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.
- The memory model for a process and its associated threads are given in the figure.

Threads (continued)

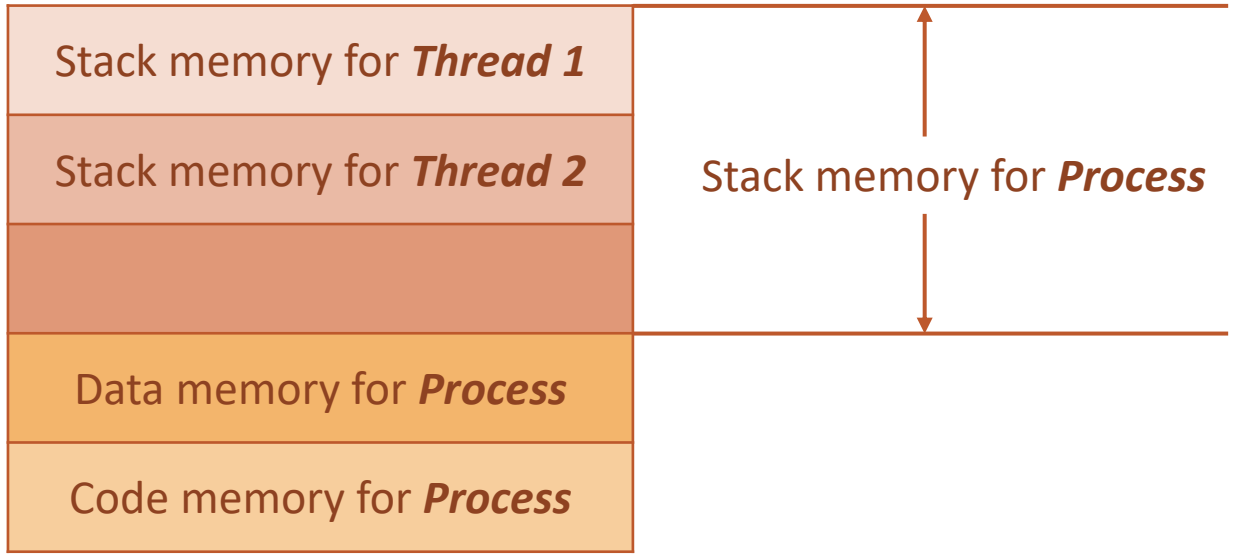


Fig: Memory organisation of a ***Process*** and its associated ***Threads***

The Concept of 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 subfunctions 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.

The Concept of Multithreading (continued)

- 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.
- 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.
- The multithreaded architecture of a process can be better visualised with the thread-process diagram, shown in the figure.

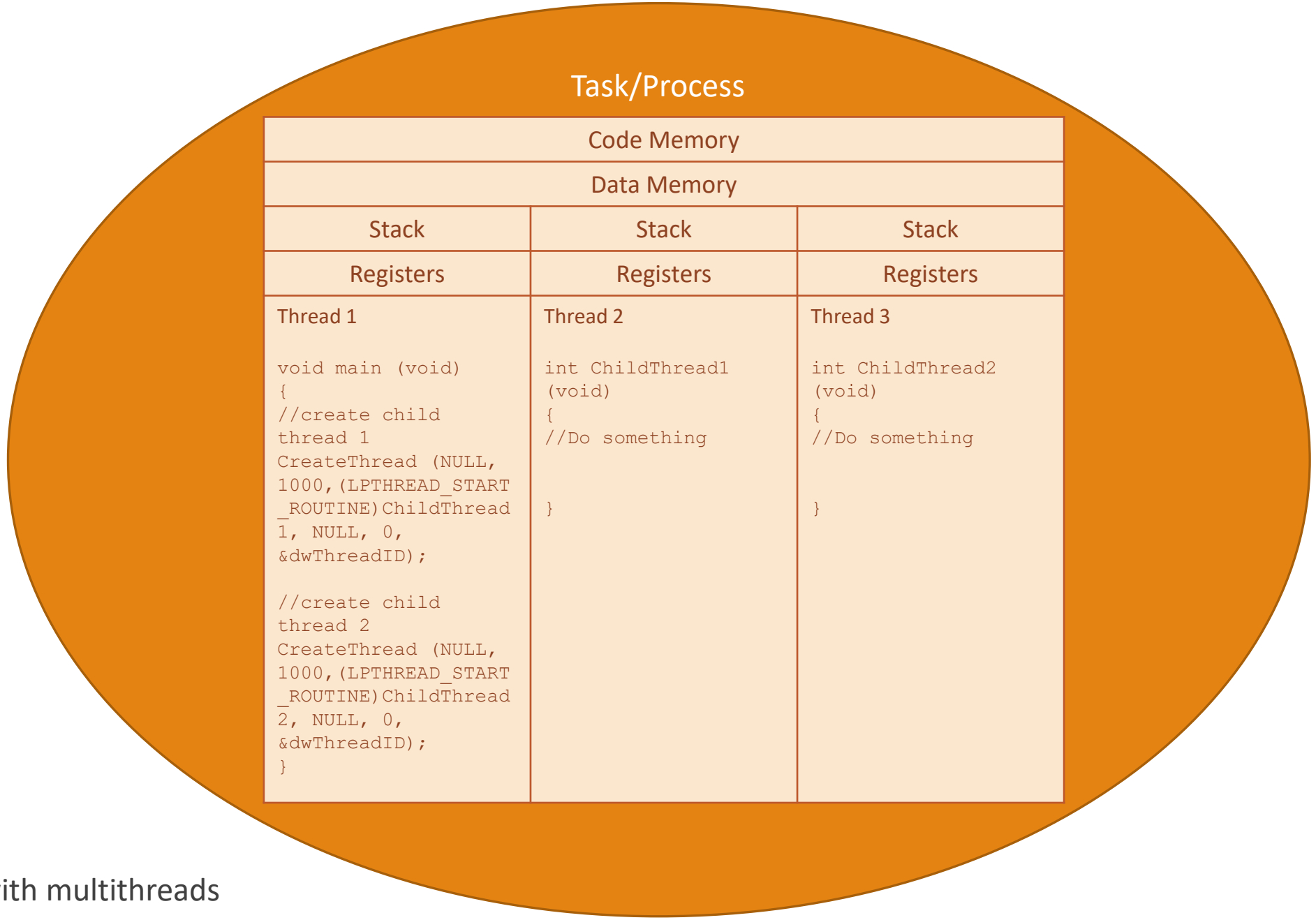


Fig: Process with multithreads

The Concept of Multithreading (continued)

- Use of multiple threads to execute a process brings the following advantages:
  - **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.
  - **Speedy execution of the process**
    - 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.
  - **Efficient CPU utilisation**
    - The CPU is engaged all time.

Thread Standards

- Thread standards deal with the different standards available for thread creation and management.
  - These standards are utilised by the operating systems for thread creation and thread management.
- It is a set of thread class libraries.
- The commonly available thread class libraries are:
  - POSIX Threads
  - Win32 Threads
  - Java Threads

POSIX Threads

- POSIX stands for Portable Operating System Interface.
- The *POSIX.4* standard deals with the Real-Time extensions and *POSIX.4a* standard deals with thread extensions.
- The POSIX standard library for thread creation and management is *'Pthreads'.*
- *'Pthreads'* library defines the set of POSIX thread creation and management functions in 'C' language.

POSIX Threads (continued)

```
int pthread_create(pthread_t *new_thread_ID, const pthread_attr_t, *attribute,
void * (*start_function) (void *), void *arguments);
```

- This primitive creates a new thread for running the function *start_function*.
- Here *pthread_t* is the handle to the newly created thread and *pthread_attr_t* is the data type for holding the thread attributes.
- *'start_function'* is the function the thread is going to execute and *arguments* is the arguments for *'start_function'*.
- On successful creation of a *Pthread*, *pthread_create()* associates the Thread Control Block (TCB) corresponding to the newly created thread to the variable of type *pthread_t* (*new_thread ID* in our example).

POSIX Threads (continued)

```
int pthread_join(pthread_t new_thread, void * *thread_status);
```

- This primitive blocks the current thread and waits until the completion of the thread pointed by it (*new_thread* in this example).
- All the POSIX 'thread calls' returns an integer.
  - A return value of zero indicates the success of the call.

POSIX Threads - Example

- Write a multithreaded application to print 'Hello I'm in main thread" from the main thread and "Hello I'm in new thread" 5 times each, using the *pthread_create()* and *pthread_join()* POSIX primitives.

```
//Assumes the application is running on an OS where POSIX library is available
#include<pthread.h>
#include<stdlib.h>
#include<stdio.h>
//*************************************************************************
//New thread function for printing “Hello I’m in new thread”
void *new_thread(void *thread_args)
{
 int i,j;
 for(j=0; j<5; j++)
 {
  printf(“Hello I’m in new thread\n”);
  for(i=0; i<10000; i++);            //Wait for some time. Do nothing.
 }
 return NULL;
}
```

```
//*************************************************************************
//Start of main thread
int main (void)
{
 int i,j;
 pthread_t tcb;
//Create the new thread for executing new_thread function
 if (pthread_create(&tcb, NULL, new_thread, NULL))
 {
  //New thread creation failed
  printf("Error in creating new thread\n");
  return -1;
 }
 for(j=0; j<5; j++)
 {
  printf("Hello I'm in main thread\n");
  for(i=0; i<10000; i++);              //Wait for some time. Do nothing.
 }
 if (pthread_join(tcb, NULL))
 {
  //Thread join failed
  printf("Error in Thread join\n");
  return -1;
 }
return 1;
}
```

POSIX Threads (continued)

- The termination of a thread can happen in different ways:
  - Natural termination:
    - The thread completes its execution and returns to the main thread through a simple *return* or by executing the *pthread_exit()* call.
  - Forced termination:
    - This can be achieved by the call *pthread_cancel()* or through the termination of the main thread with *exit* or *exec* functions.
    - *pthread_cancel()* call is used by a thread to terminate another thread.

Thread Pre-emption

- Thread pre-emption is the act of pre-empting the currently running thread.
  - It means, stopping the currently running thread temporarily.
- Thread pre-emption is performed for sharing the CPU time among all the threads.
- The execution switching among threads is known as 'Thread context switching’.
- Thread context switching is dependent on the Operating system's scheduler and the type of the thread.

Types of Threads

- User Level Threads
  - User level threads do not have kernel/Operating System support and they exist solely in the running process.
  - Even if a process contains multiple user level threads, the OS treats it as single thread and will not switch the execution among the different threads of it.
    - It is the responsibility of the process to schedule each thread as and when required.
  - In summary, user level threads of a process are non-preemptive at thread level from OS perspective.
  - The execution switching (thread context switching) happens only when the currently executing user level thread is voluntarily blocked.
    - Hence, no OS intervention and system calls are involved in the context switching of user level threads.
    - This makes context switching of user level threads very fast.

Types of Threads (continued)

- Kernel Level Threads
  - Kernel level threads are individual units of execution, which the OS treats as separate threads.
  - The OS interrupts the execution of the currently running kernel thread and switches the execution to another kernel thread based on the scheduling policies implemented by the OS.
  - In summary, kernel level threads are pre-emptive.
  - Kernel level threads involve lots of kernel overhead and involve system calls for context switching.
  - However, kernel threads maintain a clear layer of abstraction and allow threads to use system calls independently.

Thread Binding Models

- There are many ways for binding user level threads with system/kernel level threads.
- ***Many-to-One Model***
  - Here, many user level threads are mapped to a single kernel thread.
  - In this model, the kernel treats all user level threads as single thread and the execution switching among the user level threads happens when a currently executing user level thread voluntarily blocks itself or relinquishes the CPU.
  - Solaris Green threads and GNU Portable Threads are examples for this.
  - The *'PThread' example* is an illustrative example for application with Many-to-One thread model.

Thread Binding Models (continued)

- ***One-to-One Model***
  - Here, each user level thread is bonded to a kernel/system level thread.
  - Windows XP/NT/2000 and Linux threads are examples for One-to-One thread models.
  - The modified *'PThread'* example is an illustrative example for application with One-to-One thread model.
- ***Many-to-Many Model***
  - In this model, many user level threads are allowed to be mapped to many kernel threads.
  - Windows NT/2000 with *ThreadFibre* package is an example for this.

Thread vs. Process

Thread	Process
Thread is a single unit of execution and is part of process.	Process is a program in execution and contains one or more threads.
A thread does not have its own data memory and heal memory. It shares the data memory and heap memory with other threads of the same process.	Process has its own code memory, data memory and stack memory.
A thread cannot live independently; it lives within the process.	A process contains at least one thread.
There can be multiple threads in a process. The first thread (main thread) calls the main function and occupies the start of the stack memory of the process.	Threads within a process share the code, data and heap memory. Each thread holds separate memory area for stack (share the total stack memory of the process).
Threads are very inexpensive to create.	Processes are very expensive to create. Involves many OS overhead.
Context switching is inexpensive and fast.	Context switching is complex and involves lot of OS overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also die.

Task Scheduling

Task Scheduling

- Multitasking involves the execution switching among the different tasks.
- There should be some mechanism in place to share the CPU among the different tasks and to decide which process/task is to be executed at a given point of time.
- Determining which task/process is to be executed at a given point of time is known as *task/process scheduling*.
- Scheduling policies forms the guidelines for determining which task is to be executed when.
- The scheduling policies are implemented in an algorithm and it is run by the kernel as a service.
- The kernel service/application, which implements the scheduling algorithm, is known as *'Scheduler'*.

Task Scheduling

- Based on the scheduling algorithm used, scheduling can be classified into:
  - Non-preemptive Scheduling
    - The currently executing task/process is allowed to run until it terminates or enters the '*Wait*' state waiting for an I/O or system resource.
  - Preemptive Scheduling
    - The currently executing task/process is preempted (stopped temporarily) and another task from the Ready queue is selected for execution.

Task Scheduling (continued)

- The process scheduling decision may take place when a process switches its state to
  1. '*Ready*' state from '*Running*' state
  2. '*Blocked/Wait*' state from '*Running*' state
  3. '*Ready*' state from '*Blocked/Wait*' state
  4. '*Completed*' state
- A process switches to 'Ready' state from the 'Running' state when it is preempted.
  - Hence, the type of scheduling in scenario 1 is pre-emptive.

Task Scheduling (continued)

- When a high priority process in the 'Blocked/Wait' state completes its I/O and switches to the 'Ready' state, the scheduler picks it for execution if the scheduling policy used is priority based preemptive.
  - This is indicated by scenario 3.
- In preemptive/non-preemptive multitasking, the process relinquishes the CPU when it enters the ‘Blocked/Wait' state or the 'Completed' state and switching of the CPU happens at this stage.
- Scheduling under scenario 2 can be either preemptive or non-preemptive.
- Scheduling under scenario 4 can be preemptive, non-preemptive or co-operative.

Task Scheduling (continued)

- The selection of a scheduling criterion/algorithm should consider the following factors:
  - ***CPU Utilisation:***
    - The scheduling algorithm should always make the CPU utilisation high.
    - CPU utilisation is a direct measure of how much percentage of the CPU is being utilised.
  - ***Throughput:***
    - This gives an indication of the number of processes executed per unit of time.
    - The throughput for a good scheduler should always be higher.
  - ***Turnaround Time (TAT):***
    - It is the amount of time taken by a process for completing its execution.
    - It includes the time spent by the process for waiting for the main memory, time spent in the ready queue, time spent on completing the I/O operations, and the time spent in execution.
    - The turnaround time should be minimal for a good scheduling algorithm.

Task Scheduling (continued)

- ***Waiting Time:***
  - It is the amount of time spent by a process in the 'Ready' queue waiting to get the CPU time for execution.
  - The waiting time should be minimal for a good scheduling algorithm.
- ***Response Time:***
  - It is the time elapsed between the submission of a process and the first response.
  - For a good scheduling algorithm, the response time should be as least as possible.

To summarise, a good scheduling algorithm has high CPU utilisation, minimum Turn Around Time (TAT), maximum throughput and least response time.

Task Scheduling (continued)

- The various queues maintained by OS in association with CPU scheduling are:
  - ***Job Queue:***
    - Contains all the processes in the system.
  - ***Ready Queue:***
    - Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution.
    - The Ready queue is empty when there is no process ready for running.
  - ***Device Queue:***
    - Contains the set of processes, which are waiting for an I/O device.

Preemptive Scheduling

- In preemptive scheduling, the scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the 'Ready' queue for execution.
  - Every task in the 'Ready' queue gets a chance to execute.
- When to pre-empt a task and which task is to be picked up from the 'Ready' queue for execution after preempting the current task is purely dependent on the scheduling algorithm.
- A task which is preempted by the scheduler is moved to the 'Ready' queue.
- The act of moving a 'Running' process/task into the 'Ready' queue by the scheduler, without the processes requesting for it is known as ‘Preemption’

Preemptive Scheduling Techniques

- Preemptive scheduling can be implemented in different approaches.
  - Time-based preemption
  - Priority-based preemption
- The various types of preemptive scheduling adopted in task/process scheduling are:
  - Preemptive Shortest Job First (SJF)/Shortest Remaining Time (SRT) Scheduling
  - Round Robin (RR) Scheduling
  - Priority Based Scheduling

Preemptive Shortest Job First (SJF)/Shortest Remaining Time (SRT) Scheduling

- In SJF, the process with the shortest estimated run time is scheduled first, followed by the next shortest process, and so on.
- The preemptive SJF scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated time for the currently executing process.
- If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution.
- Thus preemptive SJF scheduling always compares the execution completion time (It is same as the remaining time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution.
  - Preemptive SJF scheduling is also known as Shortest Remaining Time (SRT) scheduling .

Preemptive SJF/SRT Scheduling - Example

- Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enter the ready queue together. A new process P4 with estimated completion time 2 ms enters the 'Ready' queue after 2 ms. Assume all the processes contain only CPU operation and no I/O operations are involved. Calculate the waiting time and Turn Around Time (TAT) for each process and the average waiting time and Turn Around Time in the SRT scheduling.

Preemptive SJF/SRT Scheduling – Example (continued)

Process ID	Remaining Time
P1	10 ms
P2	5 ms
P3	7 ms

'Ready' queue at 0 ms

P2 is scheduled

Process ID	Remaining Time
P1	10 ms
P2	3 ms
P3	7 ms
P4	2 ms

'Ready' queue at 2 ms

P2 is preempted
P4 is scheduled

Process ID	Remaining Time
P1	10 ms
P2	3 ms
P3	7 ms

'Ready' queue at 4 ms

P4 is completed
P2 is scheduled

Process ID	Remaining Time
P1	10 ms
P3	7 ms

'Ready' queue at 7 ms

P2 is completed
P3 is scheduled

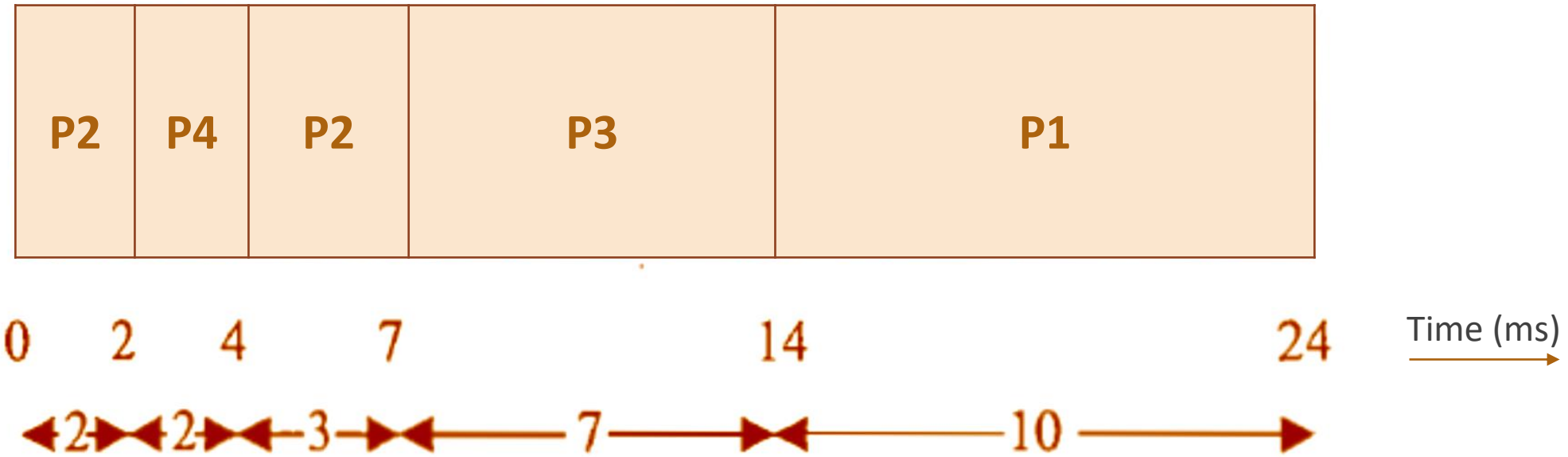
Process ID	Remaining Time
P1	10 ms

'Ready' queue at 14 ms

P3 is completed
P1 is scheduled

Preemptive SJF/SRT Scheduling – Example (continued)

- The execution sequence can be written as below:



Preemptive SJF/SRT Scheduling – Example (continued)

- The waiting time for all the processes are given as
  - $Waiting\ time\ for\ P2 = 0\ ms + (4\ -\ 2)\ ms = 2\ ms$
  - $Waiting\ time\ for\ P4 = 0\ ms$
  - $Waiting\ time\ for\ P3 = 7\ ms$
  - $Waiting\ time\ for\ P1 = 14\ ms$
- $$Average\ Waiting\ time = \frac{Waiting\ time\ for\ all\ the\ processes}{Number\ of\ processes}$$

$$= \frac{2+0+7+14}{4}\ ms = \frac{23}{4} ms$$

$$= 5.75\ ms$$

Preemptive SJF/SRT Scheduling – Example (continued)

- $Turn\ Around\ Time\ (TAT) = Time\ spent\ in\ ready\ queue\ + Execution\ Time$
  - $Turn\ Around\ Time\ (TAT)\ for\ P2 = 2\ ms + 5\ ms = 7\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P4 = 0\ ms + 2\ ms = 2\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P3 = 7\ ms + 7\ ms = 14\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P1 = 14\ ms + 10\ ms = 24\ ms$
- $Average\ Turn\ Around\ Time\ (TAT) = \frac{TAT\ for\ all\ the\ processes}{Number\ of\ processes}$

$$= \frac{7+2+14+24}{4}\ ms = \frac{47}{4} ms$$

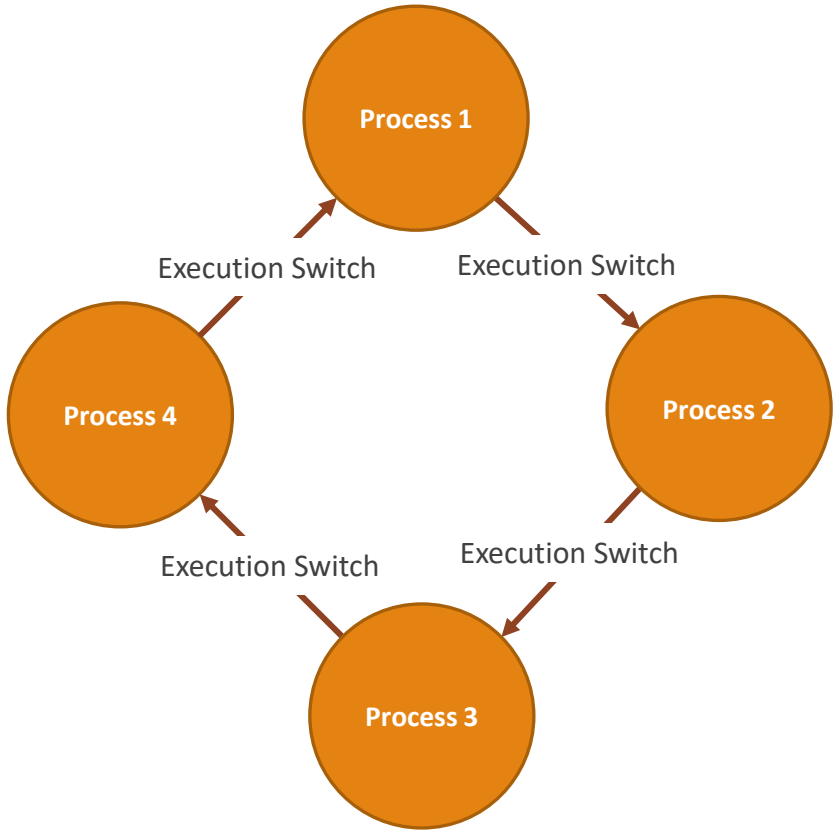
$$= 11.75\ ms$$

Round Robin (RR) Scheduling

- In Round Robin scheduling, each process in the 'Ready' queue is executed for a pre-defined time slot.
  - 'Round Robin' brings the message "Equal chance to all".
- The execution starts with picking up the first process in the 'Ready' queue.
- It is executed for a pre-defined time and when the pre-defined time elapses or the process completes (before the pre-defined time slice), the next process in the 'Ready' queue is selected for execution.
- This is repeated for all the processes in the 'Ready' queue.
- Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the 'Ready' queue again for execution.
- The sequence is repeated.

Round Robin (RR) Scheduling (continued)

- The 'Ready' queue can be considered as a circular queue in which the scheduler picks up the first process for execution and moves to the next till the end of the queue and then comes back to the beginning of the queue to pick up the first process.



Round Robin (RR) Scheduling (continued)

- The time slice is provided by the timer tick feature of the time management unit of the OS kernel.
- Time slice is kernel dependent and it varies in the order of a few microseconds to milliseconds.
- Round Robin scheduling ensures that every process gets a fixed amount of-CPU time for execution.
- When the process gets its fixed time for execution is determined by the First Come First Serve (FCFS) policy.
- If a process terminates before the elapse of the time slice, the process releases the CPU voluntarily and the next process in the queue is scheduled for execution by the scheduler.

Round Robin (RR) Scheduling - Example

- Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively, enter the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in RR algorithm with Time slice = 2 ms.

Process ID	Remaining Time
P1	6 ms
P2	4 ms
P3	2 ms

'Ready' queue at 0 ms

P1 is scheduled

Process ID	Remaining Time
P2	4 ms
P3	2 ms
P1	4 ms

'Ready' queue at 2 ms

P1 is preempted
P2 is scheduled

Process ID	Remaining Time
P3	2 ms
P1	4 ms
P2	2 ms

'Ready' queue at 4 ms

P2 is preempted
P3 is scheduled

Process ID	Remaining Time
P1	4 ms
P2	2 ms

'Ready' queue at 6 ms

P3 is completed
P1 is scheduled

Process ID	Remaining Time
P2	2 ms
P1	2 ms

'Ready' queue at 8 ms

P1 is preempted
P2 is scheduled

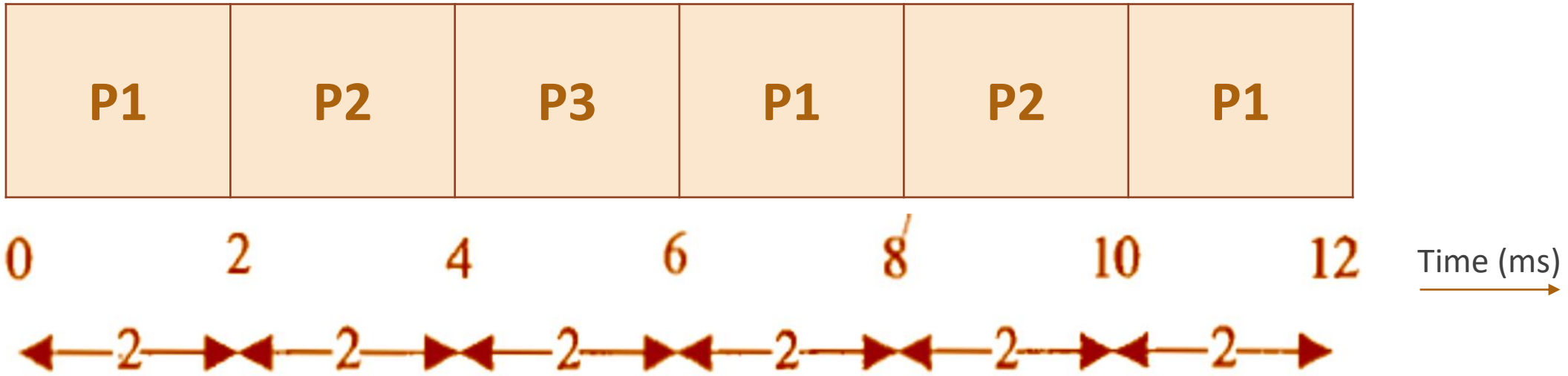
Process ID	Remaining Time
P1	2 ms

'Ready' queue at 10 ms

P2 is completed
P1 is scheduled

Round Robin (RR) Scheduling– Example (continued)

- The execution sequence can be written as below:



Round Robin (RR) Scheduling – Example (continued)

- The waiting time for all the processes are given as
  - $Waiting\ time\ for\ P1 = 0\ ms + (6\ -\ 2) + (10 - 8)\ ms = 6\ ms$
  - $Waiting\ time\ for\ P2 = (2 - 0) + (8 - 4)\ ms = 6\ ms$
  - $Waiting\ time\ for\ P3 = (4 - 0) = 4\ ms$
- $$Average\ Waiting\ time = \frac{Waiting\ time\ for\ all\ the\ processes}{Number\ of\ processes}$$
$$= \frac{6+6+4}{3}\ ms = \frac{16}{3}\ ms$$
$$= 5.33\ ms$$

Round Robin (RR) Scheduling – Example (continued)

- $Turn\ Around\ Time\ (TAT) = Time\ spent\ in\ ready\ queue\ + Execution\ Time$
  - $Turn\ Around\ Time\ (TAT)\ for\ P1 = 6\ ms + 6\ ms = 12\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P2 = 6\ ms + 4\ ms = 10\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P3 = 4\ ms + 2\ ms = 6\ ms$
- $Average\ Turn\ Around\ Time\ (TAT) = \frac{TAT\ for\ all\ the\ processes}{Number\ of\ processes}$

$$= \frac{12+10+6}{3}\ ms = \frac{28}{3}\ ms$$

$$= 9.33\ ms$$

Priority Based Scheduling

- The Priority Based Preemptive Scheduling ensures that a process with high priority is serviced at the earliest compared to other low priority processes in the ‘Ready’ queue.
  - Any high priority process entering the 'Ready' queue is immediately scheduled for execution.
- The priority of a task/process can be indicated through various mechanisms.
  - While creating the process/task, the priority can be assigned to it.
  - The priority number associated with a task/process is the direct indication of its priority.
  - The priority number 0 indicates the highest priority.
  - This convention need not be universal and it depends on the kernel level implementation of the priority structure.
- Whenever a new process enters the ‘Ready’ queue, the scheduler sorts the 'Ready' queue based on priority and picks the process with the highest level of priority for execution.

Priority Based Scheduling - Example

- Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0 – highest priority, 3 - lowest priority) respectively enter the ready queue together. A new process P4 with estimated completion time 6 ms and priority 0 enters the 'Ready' queue after 5 ms of start of execution of P1. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in priority based scheduling algorithm.

Priority	Process ID	Remaining Time
1	P1	10 ms
3	P2	5 ms
2	P3	7 ms

'Ready' queue at 0 ms

P1 is scheduled

Priority	Process ID	Remaining Time
1	P1	5 ms
3	P2	5 ms
2	P3	7 ms
0	P4	6 ms

'Ready' queue at 5 ms

P1 is preempted
P4 is scheduled

Priority	Process ID	Remaining Time
1	P1	5 ms
3	P2	5 ms
2	P3	7 ms

'Ready' queue at 11 ms

P4 is completed
P1 is scheduled

Priority	Process ID	Remaining Time
3	P2	5 ms
2	P3	7 ms

'Ready' queue at 16 ms

P1 is completed
P3 is scheduled

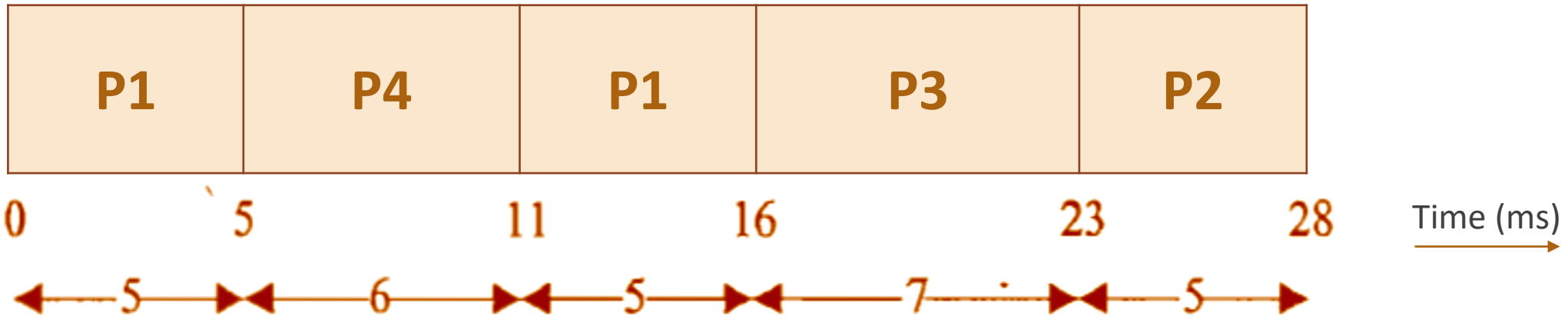
Priority	Process ID	Remaining Time
3	P2	5 ms

'Ready' queue at 23 ms

P3 is completed
P2 is scheduled

Priority Based Scheduling– Example (continued)

- The execution sequence can be written as below:



Priority Based Scheduling – Example (continued)

- The waiting time for all the processes are given as
  - $Waiting\ time\ for\ P1 = 0\ ms + (11 - 5)\ ms = 6\ ms$
  - $Waiting\ time\ for\ P4 = 0\ ms$
  - $Waiting\ time\ for\ P3 = 16\ ms$
  - $Waiting\ time\ for\ P2 = 23\ ms$
- $$Average\ Waiting\ time = \frac{Waiting\ time\ for\ all\ the\ processes}{Number\ of\ processes}$$

$$= \frac{6+0+16+23}{4}\ ms = \frac{45}{4}\ ms$$

$$= 11.25\ ms$$

Priority Based Scheduling – Example (continued)

- $Turn\ Around\ Time\ (TAT) = Time\ spent\ in\ ready\ queue\ + Execution\ Time$
  - $Turn\ Around\ Time\ (TAT)\ for\ P1 = 6\ ms + 10\ ms = 16\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P4 = 0\ ms + 6\ ms = 6\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P3 = 16\ ms + 7\ ms = 23\ ms$
  - $Turn\ Around\ Time\ (TAT)\ for\ P2 = 23\ ms + 5\ ms = 28\ ms$
- $Average\ Turn\ Around\ Time\ (TAT) = \frac{TAT\ for\ all\ the\ processes}{Number\ of\ processes}$

$$= \frac{16+6+23+28}{4}\ ms = \frac{73}{4}\ ms$$

$$= 18.25\ ms$$

Task Communication

Task Communication

- In a multitasking system, multiple tasks/processes run concurrently (in pseudo parallelism) and each process may or may not interact between.
- Based on the degree of interaction, the processes running on an OS are classified as
  - **Co-operating Processes:**
    - One process requires the inputs from other processes to complete its execution.
  - **Competing Processes:**
    - The competing processes do not share anything among themselves but they share the system resources.
    - The competing processes compete for the system resources such as file, display device, etc.

Task Communication (continued)

- Co-operating processes exchanges information and communicate through the following methods:
  - **Co-operation through Sharing:**
    - The co-operating process exchange data through some shared resources.
  - **Co-operation through Communication:**
    - No data is shared between the processes.
    - But they communicate for synchronisation.

Task Communication (continued)

- The mechanism through which processes/tasks communicate each other is known as Inter Process/Task Communication (IPC).
  - Inter Process Communication is essential for process co-ordination.
- The various types of Inter Process Communication (IPC) mechanisms adopted by process are kernel (Operating System) dependent.
- Some of the important IPC mechanisms adopted by various kernels are:
  - ***Shared Memory***
    - Pipes and Memory Mapped Objects
  - ***Message Passing***
    - Message Queue, Mailbox and Signalling
  - ***Remote Procedure Call and Sockets***

Shared Memory

- Processes share some area of the memory to communicate among them.
- Information to be communicated by the process is written to the shared memory area.
- Other processes which require this information can read the same from the shared memory area.

Fig.: Concept of Shared Memory

- Different mechanisms are adopted by different kernels for implementing the concept of shared memory:
  - Pipes
  - Memory Mapped Objects

Pipes

- *'Pipe'* is a section of the shared memory used by processes for communicating.
- Pipes follow the client-server architecture.
  - A process which creates a pipe is known as a *pipe server* and a process which connects to a pipe is known as *pipe client*.
- A pipe can be considered as a conduit for information flow and has two conceptual ends.
- It can be unidirectional, allowing information flow in one direction or bidirectional allowing bidirectional information flow.

Pipes (continued)

- A unidirectional pipe allows the process connecting at one end of the pipe to write to the pipe and the process connected at the other end of the pipe to read the data, whereas a bidirectional pipe allows both reading and writing at one end.
- The figure shows a unidirectional pipe.

Fig.: Concept of Pipe for IPC

Pipes (continued)

- The implementation of 'Pipes' is OS dependent.
- Microsoft Windows supports two types of 'Pipes' for Inter Process Communication:
  - **Anonymous Pipes:**
    - The anonymous pipes are unnamed, unidirectional pipes used for data transfer between two processes.
  - **Named Pipes:**
    - Named pipe is a named, unidirectional or bi-directional pipe for data exchange between processes.
    - Like anonymous pipes, the process which creates the named pipe is known as pipe server and a process which connects to the named pipe is known as pipe client.
    - With named pipes, any process can act as both client and server allowing point-to-point communication.
    - Named pipes can be used for communicating between processes running on the same machine or between processes running on different machines connected to a network.

Memory Mapped Objects

- Memory mapped object is a shared memory technique adopted by certain Real-Time Operating Systems for allocating a shared block of memory which can be accessed by multiple process simultaneously.
- In this approach, a mapping object is created and physical storage for it is reserved and committed.
- A process can map the entire committed physical area or a block of it to its virtual address space.
- All read and write operation to this virtual address space by a process is directed to its committed physical area.
- Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data.

Message Passing

- Message passing is an (a)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 information/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 Queue
  - Mailbox
  - Signalling

Message Queue

- 'Message queue' is a First-In-First-Out (FIFO) queue which stores the messages temporarily in a system defined memory object to pass it to the desired process.
- Usually the process which wants to talk to another process posts the message to a message queue.
- Messages are sent and received through *send* and *receive* methods.
  - *send (Name of the process to which the message is to be sent, message)*
  - *receive (Name of the process from which the message is to be received, message)*
- The implementation of the message queue, send and receive methods are OS kernel dependent.

Message Queue (continued)

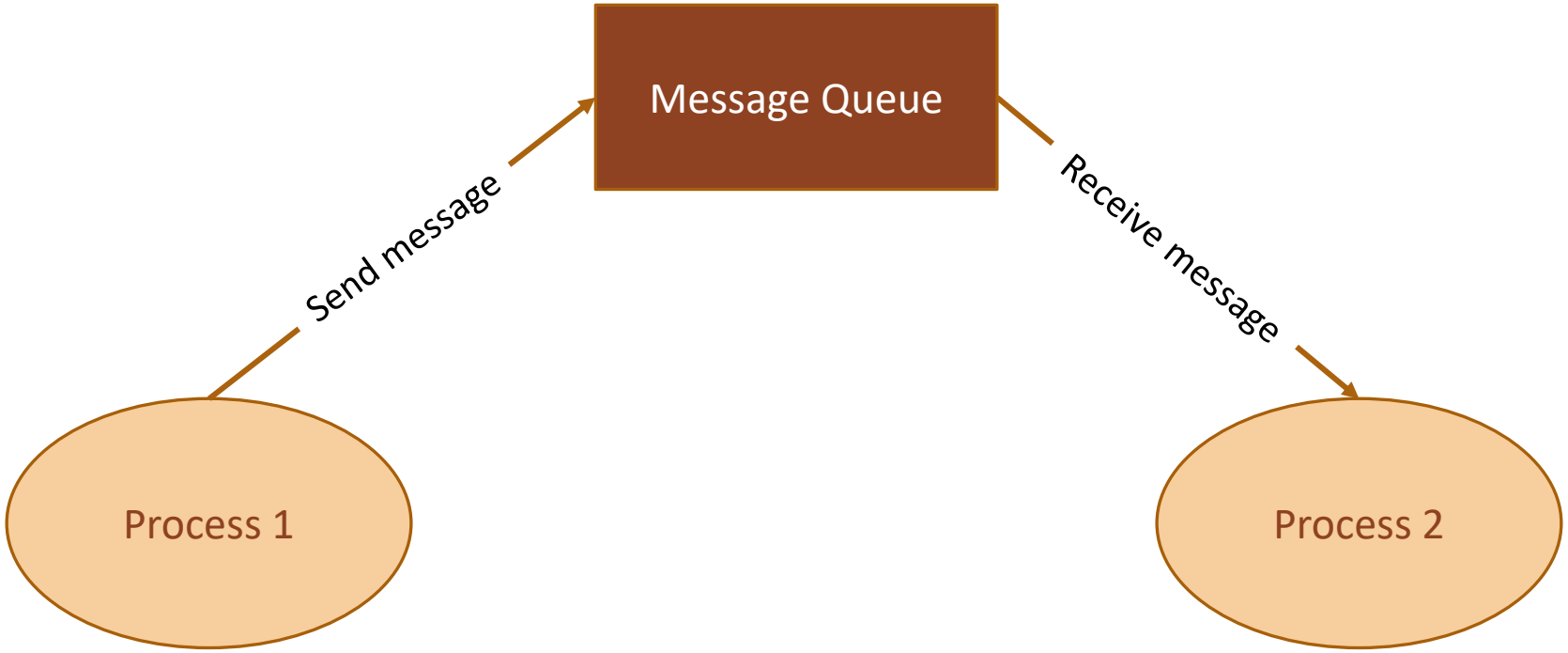


Fig.: Concept of Message Queue based indirect messaging for IPC

Message Queue (continued)

- The Windows XP OS kernel maintains a single system message queue and one process/thread specific message queue.
- A thread which wants to communicate with another thread posts the message to the system message queue.
- The kernel picks up the message from the system message queue one at a time and examines the message for finding the destination thread and then posts the message to the message queue of the corresponding thread.
- The messaging mechanism is classified into synchronous and asynchronous based on the behaviour of the message posting thread.
  - In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance (return) from the thread to which the message is posted.
  - In synchronous messaging, the thread which posts a message enters waiting state and waits for the message result from the thread to which the message is posted.
    - The thread which invoked the send message becomes blocked and the scheduler will not pick it up for scheduling.

Mailbox

- Mailbox is an alternate form of ‘Message queue’ and it is used in RTOS for IPC usually for one way messaging.
- The task/thread which wants to send a message to other tasks/threads creates a mailbox for posting the messages.
- The threads which are interested in receiving the messages posted to the mailbox by the mailbox creator thread can subscribe to the mailbox.
- The thread which creates the mailbox is known as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients’.
  - The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox.
  - The clients read the message from the mailbox on receiving the notification.

Mailbox (continued)

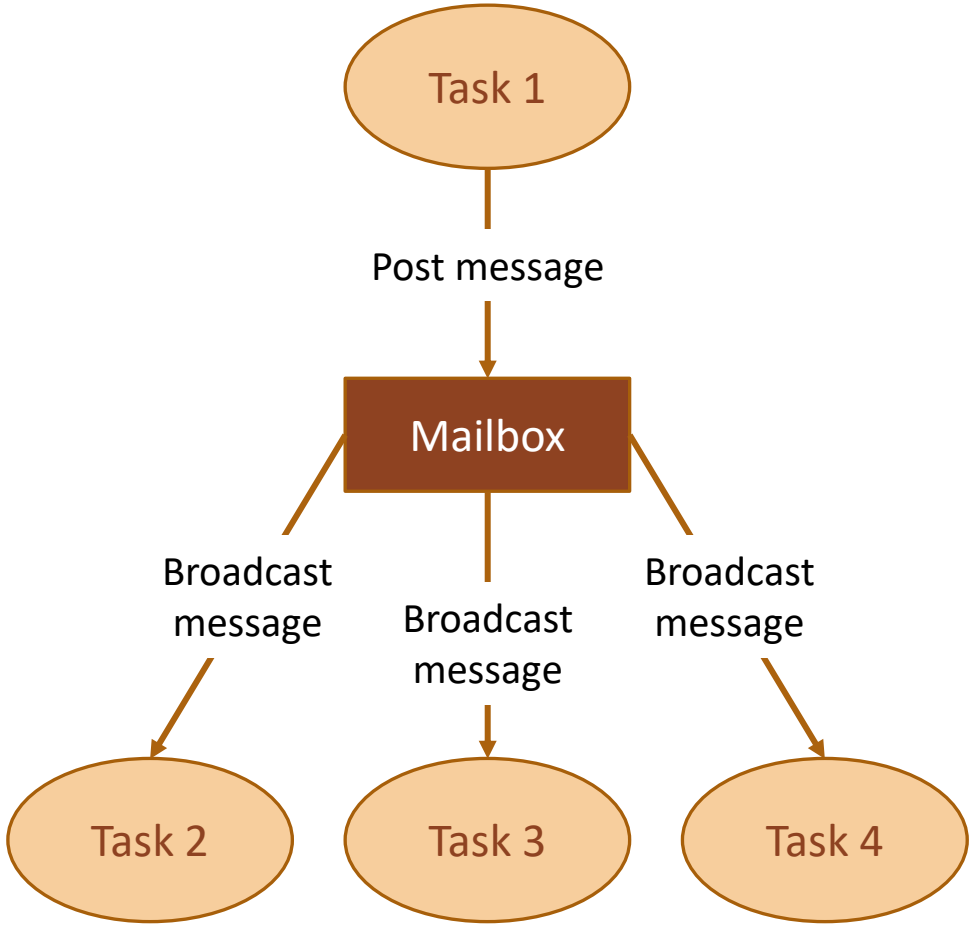


Fig.: Concept of Mailbox based indirect messaging for IPC

Mailbox (continued)

- The mailbox creation, subscription, message reading and writing are achieved through OS kernel provided API calls.
- Mailbox and message queues are same in functionality.
  - The only difference is in the number of messages supported by them.
  - Both of them are used for passing data in the form of message(s) from a task to another task(s).
  - Mailbox is used for exchanging a single message between two tasks or between an Interrupt Service Routine (ISR) and a task.
  - Mailbox associates a pointer pointing to the mailbox and a wait list to hold the tasks waiting for a message to appear in the mailbox.

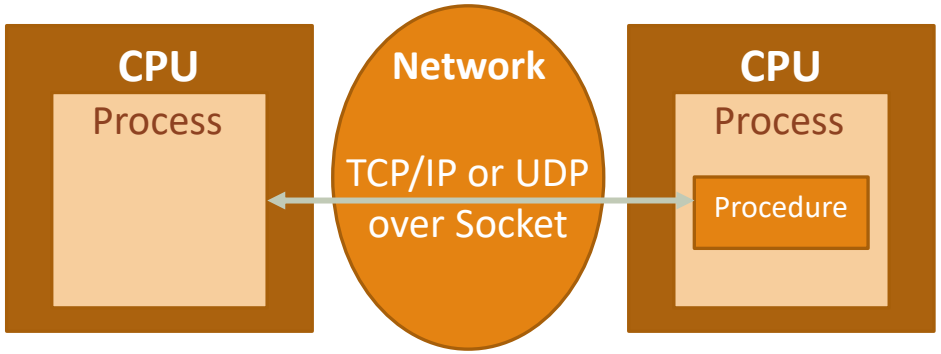
Signalling

- Signalling is a primitive way of communication between processes/threads.
- Signals are used for asynchronous notifications where one process/thread fires a signal, indicating the occurrence of a scenario which the other process(es)/thread(s) is waiting.
- Signals are not queued and they do not carry any data.
- E.g. Communication mechanisms used in RTX51 Tiny OS, inter process communication in VxWorks OS Kernel are examples for signalling.

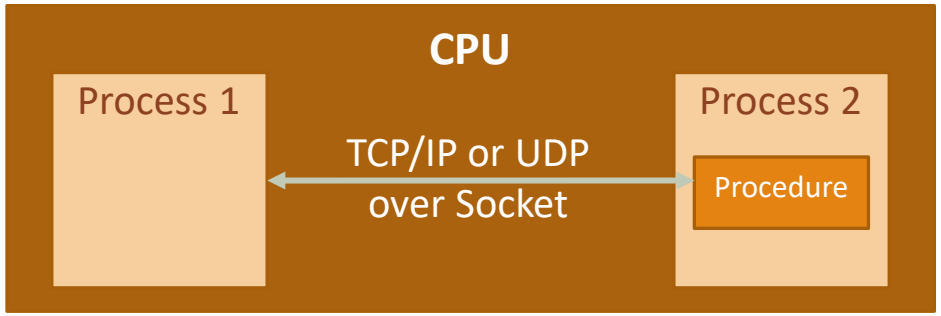
Remote Procedure Call (RPC) and Sockets

- Remote Procedure Call (RPC) is the Inter Process Communication (IPC) mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network.
- In the object oriented language terminology, RPC is also known as *Remote Invocation* or *Remote Method Invocation (RMI)*.
- RPC is mainly used for distributed applications like client-server applications.
  - With RPC it is possible to communicate over a heterogeneous network (i.e. Network where Client and server applications are running on different operating systems).
  - The CPU/process containing the procedure which needs to be invoked remotely is known as server.
  - The CPU/process which initiates an RPC request is known as client.

Remote Procedure Call (RPC) and Sockets (continued)



Processes running on different CPUs which are networked



Processes running on the same CPU

Fig.: Concept of Remote Procedure Call (RPC) for IPC

Remote Procedure Call (RPC) and Sockets (continued)

- It is possible to implement RPC communication with different invocation interfaces.
- Interface Definition Language (IDL) defines the interfaces for RPC.
  - Microsoft Interface Definition Language (MIDL) is the IDL implementation from Microsoft for all Microsoft platforms.
- The RPC communication can be either Synchronous (Blocking) or Asynchronous (Non-blocking).
  - In the Synchronous communication, the process which calls the remote procedure is blocked until it receives a response back from the other process.
  - In asynchronous RPC calls, the calling process continues its execution while the remote process performs the execution of the procedure.
    - The result from the remote procedure is returned back to the caller through mechanisms like callback functions.

Remote Procedure Call (RPC) and Sockets (continued)

- On security front, RPC employs authentication mechanisms to protect the systems against vulnerabilities.
- The client applications (processes) should authenticate themselves with the server for getting access.
- Authentication mechanisms like IDs, public key cryptography (like DES, 3DES), etc. are used by the client for authentication.
- Without authentication, any client can access the remote procedure.
  - This may lead to potential security risks.

Remote Procedure Call (RPC) and Sockets (continued)

- Sockets are used for RPC communication.
- *Socket is a logical endpoint in a two-way communication link between two applications running on a network.*
- A port number is associated with a socket so that the network layer of the communication channel can deliver the data to the designated application.
- Sockets are of different types, namely, Internet sockets (INET), UNIX sockets, etc.
- The INET socket works on internet communication protocol.
  - TCP/IP, UDP, etc. are the communication protocols used by INET sockets.

Remote Procedure Call (RPC) and Sockets (continued)

- INET sockets are classified into:
  - **Stream sockets**
    - These are connection oriented and they use TCP to establish a reliable connection.
  - **Datagram sockets**
    - These rely on UDP for establishing a connection.
    - The UDP connection is unreliable when compared to TCP.

Remote Procedure Call (RPC) and Sockets (continued)

- The client-server communication model uses a socket at the client side and a socket at the server side.
- A port number is assigned to both of these sockets.
  - The client and server should be aware of the port number associated with the socket.
- In order to start the communication, the client needs to send a connection request to the server at the specified port number.
- The client should be aware of the name of the server along with its port number.
- The server always listens to the specified port number on the network.
- Upon receiving a connection request from the client, based on the success of authentication, the server grants the connection request and a communication channel is established between the client and server.

Remote Procedure Call (RPC) and Sockets (continued)

- The client uses the host name and port number of server for sending requests and server uses the client's name and port number for sending responses.
- If the client and server applications (both processes) are running on the same CPU, both can use the same host name and port number for communication.
- The physical communication link between the client and server uses network interfaces like Ethernet or Wi-Fi for data communication.
- The underlying implementation of socket is OS kernel dependent.
  - Different types of OSs provide different socket interfaces.

Task Synchronisation

Task Synchronisation Issues

- In a multitasking environment, multiple processes run concurrently (in pseudo parallelism) and share the system resources.
- The processes communicate with each other with different IPC mechanisms including shared memory and variables.
- Imagine a situation where two processes try to access display hardware connected to the system or two processes try to access a shared memory area where one process tries to write to a memory location when the other process is trying to read from this.
  - This would result in unexpected results.
  - This can be solved by making each process aware of the access of a shared resource either directly or indirectly.

Task Synchronisation Issues (continued)

- The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as ‘Task/Process Synchronisation’.
- Various task communication/synchronisation issues may arise in a multitasking environment if processes are not synchronised properly.
  - Racing
  - Deadlock

Racing

- Let us have a look at the following piece of code:

```
#include<windows.h>
#include<stdio.h>
//*************************************************************************
//counter is an integer variable and Buffer is a byte array
//shared between two processes Process A and Process B
char Buffer[10] = {1,2,3,4,5,6,7,8,9,10};
short int counter = 0;
//*************************************************************************
//Process A
void Process_A (void){
 int i;
 for (i=0; i<5; i++){
  if (Buffer[i]>0)
   counter++;
 }
}
```

Racing (continued)

```
//*********************************************************************
//Process B
void Process_B (void){
 int j;
 for (j=5; j<10; j++){
  if (Buffer[j]>0)
   counter++;
 }
}
//*********************************************************************
//Main Thread
int main(){
 DWORD id;
 CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Process_A,(LPVOID)0,0,&id);
 CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Process_B,(LPVOID)0,0,&id);
 Sleep(100000);
 return 0;
}
```

Racing (continued)

- From a programmer perspective, the value of counter will be 10 at the end of execution of processes A & B.
  - But in a real world execution, the result depends on the process scheduling policies adopted by the OS kernel.
- The program statement `counter++;` looks like a single statement from a high level programming language ('C' language) perspective.
  - The low level implementation of this statement is dependent on the underlying processor instruction set and the (cross) compiler in use.

Racing (continued)

- The low level implementation of the high level program statement `counter++;` under Windows XP operating system running on an Intel Centrino Duo processor is given below:

```
mov eax, dword ptr [ebp-4]     ;Load counter in Accumulator
add eax, 1                     ;Increment Accumulator by 1
mov dword ptr [ebp-4], eax     ;Store counter with Accumulator
```

- Both the processes Process A and Process B contain the program statement `counter++;`

Process A	Process B
`mov eax, dword ptr [ebp-4]`	`mov eax, dword ptr [ebp-4]`
`add eax, 1`	`add eax, 1`
`mov dword ptr [ebp-4], eax`	`mov dword ptr [ebp-4], eax`

Racing (continued)

- Imagine a situation where a process switching (context switching) happens from Process A to Process B when Process A is executing the `counter++;` statement.
- Imagine that the process switching happened at the point where Process A executed the low level instruction, '`mov eax, dword ptr [ebp-4]`' and is about to execute the next instruction '`add eax, 1`'.
- The scenario is illustrated in the figure.

Racing (continued)

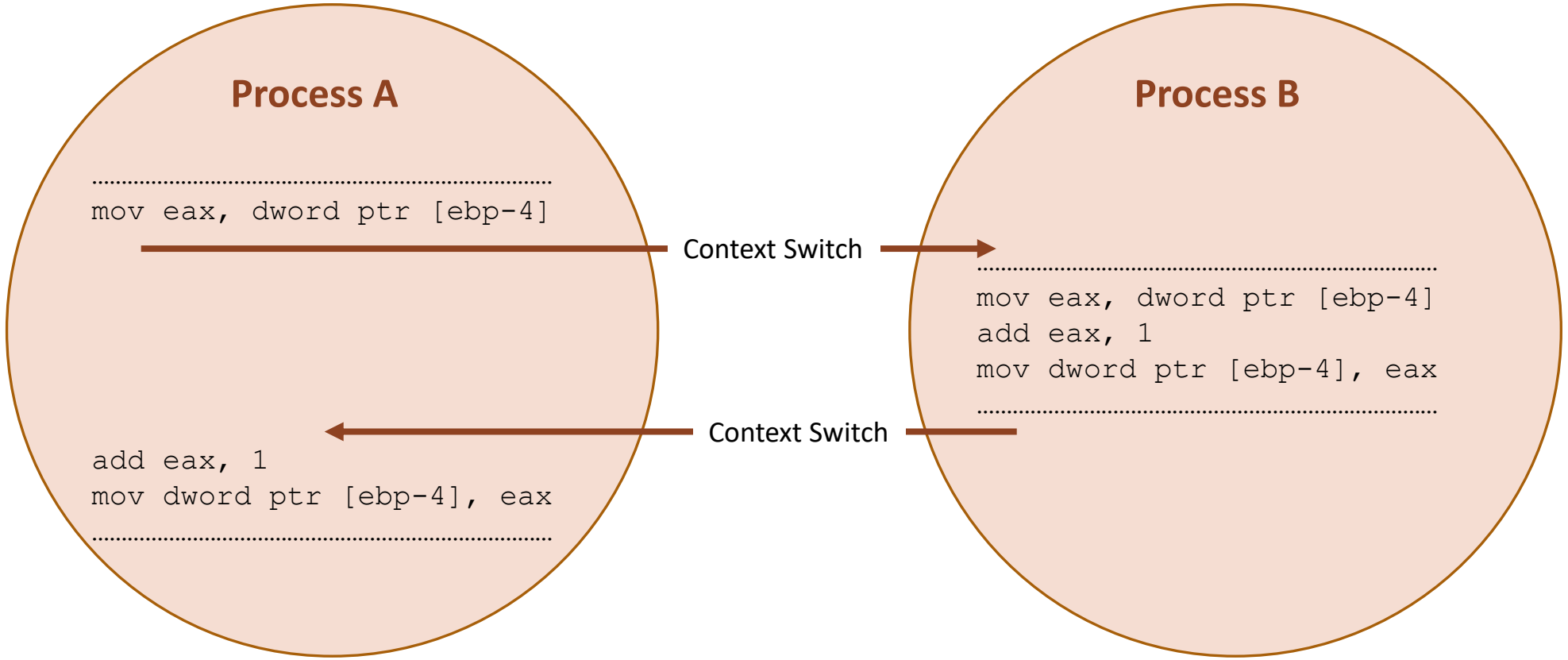


Fig.: Race Condition

Racing (continued)

- Process B increments the shared variable 'counter' in the middle of the operation where Process A tries to increment it.
- When Process A gets the CPU time for execution, it starts from the point where it got interrupted.
- Though the variable counter is incremented by Process B, Process A is unaware of it and it increments the variable with the old value.
  - This leads to the loss of one increment for the variable counter.
- This issue wouldn't have occurred if the underlying actions corresponding to the program statement `counter++;` is finished in a single CPU execution cycle.
- The best way to avoid this situation is to make the access and modification of shared variables mutually exclusive.
  - Meaning when one process accesses a shared variable, prevent the other processes from accessing it.

Racing (continued)

- To summarise, *Racing* or *Race condition* is the situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently.
- In a Race condition, the final value of the shared data depends on the process which acted on the data finally.

Deadlock

- A race condition produces incorrect results, whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes.
- This is similar to traffic jam issues in a junction as illustrated in the figure.

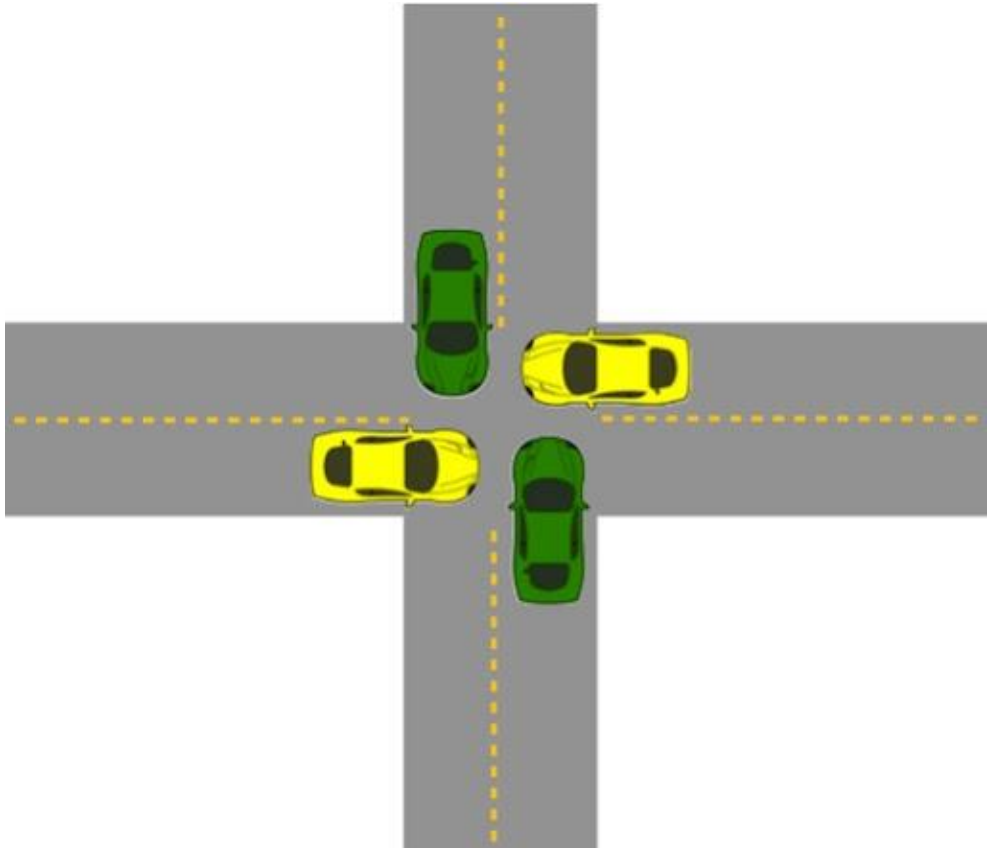


Fig.: Deadlock Visualisation

Deadlock (continued)

- In its simplest form, *deadlock* is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process.
- Process A holds a resource *x* and it wants a resource *y* held by Process B.
- Process B is currently holding resource *y* and it wants the resource *x* which is currently held by Process A.
- Both hold the respective resources and they compete each other to get the resource held by the respective processes.
- The result of the competition is 'deadlock'.
- None of the competing process will be able to access the resources held by other processes since they are locked by the respective processes.

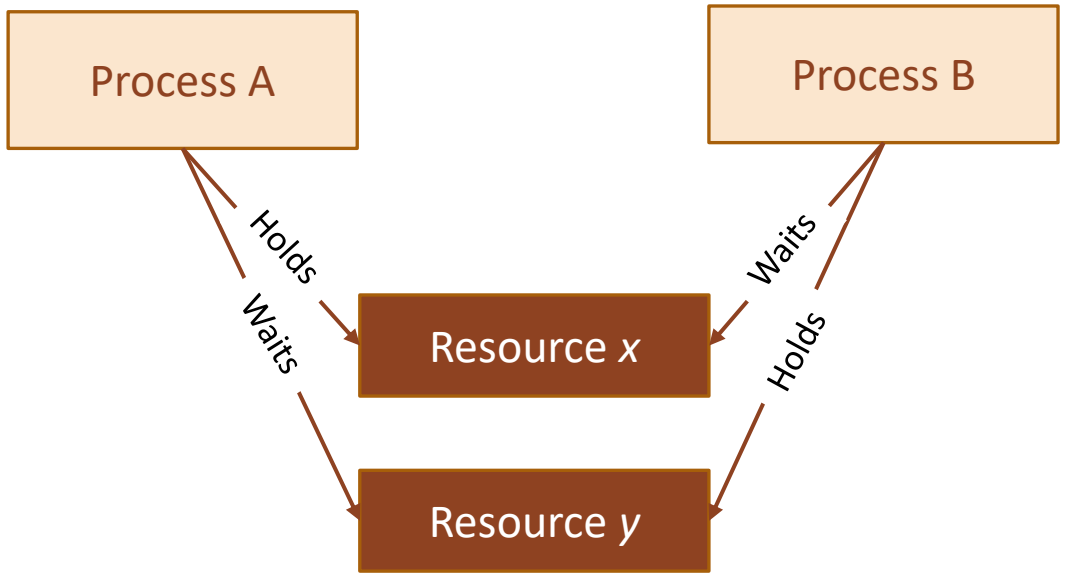


Fig.: Scenario leading to deadlock

Deadlock (continued)

- The different conditions favouring a deadlock situation are:
  - ***Mutual Exclusion:***
    - The criteria that only one process can hold a resource at a time.
    - Meaning processes should access shared resources with mutual exclusion.
    - Typical example is the accessing of display hardware in an embedded device.
  - ***Hold and Wait:***
    - The condition In which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.
  - ***No Resource Preemption:***
    - The criteria that operating system cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.

Deadlock (continued)

- ***Circular Wait:***
  - A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process.
  - In general, there exists a set of waiting process P0, P1 ... Pn with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held by P0 ,…., Pn is waiting for a resource held by P0 and P0 is waiting for a resource held by Pn and so on.
    - This forms a circular wait queue.
- 'Deadlock' is a result of the combined occurrence of these four conditions listed above.
  - These conditions were first described by E. G. Coffman in 1971 and it is popularly known as *Coffman conditions*.

Deadlock Handling

- A smart OS may foresee the deadlock condition and will act proactively to avoid such a situation.
- If a deadlock occurs, the reaction to it by OS is nonuniform.
- The OS may adopt any of the following techniques to detect and prevent deadlock conditions.
  - ***Ignore Deadlocks:***
    - Always assume that the system design is deadlock free.
    - This is acceptable for the reason that the cost of removing a deadlock is large compared to the chance of happening a deadlock.
    - UNIX is an example for an OS following this principle.
    - A life critical system cannot pretend that it is deadlock free for any reason.

Deadlock Handling (continued)

- ***Detect and Recover:***
  - This approach suggests the detection of a deadlock situation and recovery from it.
  - This is similar to the deadlock condition that may arise at a traffic junction.
  - When the vehicles from different directions compete to cross the junction, deadlock (traffic jam) condition is resulted.
  - Once a deadlock (traffic jam) has happened at the junction, the only solution is to back up the vehicles from one direction and allow the vehicles from opposite direction to cross the junction.
  - If the traffic is too high, lots of vehicles may have to be backed up to resolve the traffic jam.
  - This technique is also known as 'back up cars' technique.

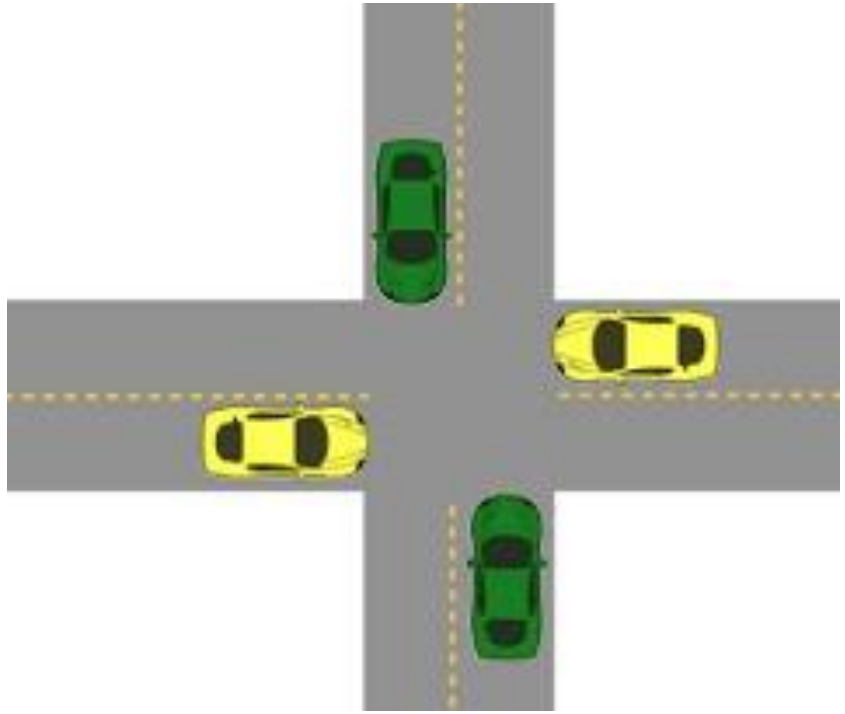


Fig.: ‘Back up cars' technique for deadline recovery

Deadlock Handling (continued)

- Operating systems keep a resource graph in their memory.
- The resource graph is updated on each resource request and release.
- A deadlock condition can be detected by analysing the resource graph by graph analyser algorithms.
- Once a deadlock condition is detected, the system can terminate a process or preempt the resource to break the deadlocking cycle.

- ***Avoid Deadlocks:***
  - Deadlock is avoided by the careful resource allocation techniques by the Operating System.
  - It is similar to the traffic light mechanism at junctions to avoid the traffic jams.
- ***Prevent Deadlocks:***
  - Prevent the deadlock condition by negating one of the four conditions favouring the deadlock situation.

Deadlock Handling (continued)

- Ensure that a process does not hold any other resources when it requests a resource.
  1. A process must request all its required resource and the resources should be allocated before the process begins its execution.
  2. Grant resource allocation requests from processes only if the process does not hold a resource currently.
- Ensure that resource preemption (resource releasing) is possible at operating system level.
  1. Release all the resources currently held by a process if a request made by the process for anew resource is not able to fulfil immediately.
  2. Add the resources which are preempted (released) to a resource list describing the resources which the process requires to complete its execution.
  3. Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

Deadlock (continued)

- **Livelock**
  - The *Livelock* condition is similar to the deadlock condition except that a process in livelock condition changes its state with time.
  - While in deadlock a process enters in wait state for a resource and continues in that state forever without making any progress in the execution, in a livelock condition a process always does something but is unable to make any progress in the execution completion.
  - The livelock condition is better explained with the real world example, two people attempting to cross each other in a narrow corridor.
    - Both the persons move towards each side of the corridor to allow the opposite person to cross.
    - Since the corridor is narrow, none of them are able to cross each other.
    - Here both of the persons perform some action but still they are unable to achieve their target, cross each other.

Deadlock (continued)

- **Starvation**
  - In the multitasking context, *starvation* is the condition in which a process does not get the resources required to continue its execution for a long time.
  - As time progresses, the process starves on resource.
  - Starvation may arise due to various conditions like byproduct of preventive measures of deadlock, scheduling policies favouring high priority tasks and tasks with shortest execution time, etc.

Task Synchronisation Techniques

- Process/Task synchronisation is essential for
  1. Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.
  2. Ensuring proper sequence of operation across processes.
  3. Communicating between processes.
- The code memory area which holds the program instructions (piece of code) for accessing a shared resource (like shared memory, shared variables, etc.) is known as *'critical section'.*
  - In order to synchronise the access to shared resources, the access to the *critical section* should be exclusive.

Task Synchronisation Techniques (continued)

- The exclusive access to critical section of code is provided through *mutual exclusion* mechanism.
- Consider two processes Process A and Process B running on a multitasking system.
- Process A is currently running and it enters its critical section.
- Before Process A completes its operation in the critical section, the scheduler preempts Process A and schedules Process B for execution (Process B is of higher priority compared to Process A).
- Process B also contains the access to the critical section which is already in use by Process A.
- If Process B continues its execution and enters the critical section which is already in use by Process A, a racing condition will be resulted.
- A mutual exclusion policy enforces mutually exclusive access of critical sections.

Task Synchronisation Techniques (continued)

- Mutual exclusion blocks a process.
- Based on the behaviour of the blocked process, mutual exclusion methods can be classified into two categories:
  - Mutual Exclusion through Busy Waiting/Spin Lock
  - Mutual Exclusion through Sleep & Wakeup

Mutual Exclusion through Sleep & Wakeup

- When a process is not allowed to access the critical section, which is currently being locked by another process, the process undergoes 'Sleep' and enters the 'blocked' state.
- The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section.
- The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when the process leaves the critical section.
- The *'Sleep & Wakeup'* policy for mutual exclusion can be implemented in different ways.
  - Windows XP/CE OS kernels use semaphores for *'Sleep & Wakeup'* policy implementation for mutual exclusion.

Semaphore

- Semaphore is a *'Sleep & Wakeup'* based mutual exclusion implementation for shared resource access.
- 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.

Semaphore (continued)

- Based on the implementation of the sharing limitation of the shared resource, semaphores are classified into two, namely *'Binary Semaphore'* and *'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.
  - Under certain OS kernel, it is referred as *mutex*.
- The *Counting Semaphore* limits the access of resources by a fixed number of processes/threads.
  - *Counting Semaphore* maintains a count between zero and a value.
  - It limits the usage of the resource to the maximum value of the count supported by it.

Counting Semaphore

- The *Counting Semaphore* limits the access of resources by a fixed number of processes/threads.
- *Counting Semaphore* maintains a count between zero and a value.
- It limits the usage of the resource to the maximum value of the count supported by it.
- The state of the counting semaphore object is set to 'signalled' when the count of the object is greater than zero.
- The count associated with a *'Semaphore object'* is decremented by one when a process/thread acquires it and the count is incremented by one when a process/thread releases the *'Semaphore object'.*
- The state of the *'Semaphore object'* is set to 'non-signalled' when the semaphore is acquired by the maximum number of processes/threads that the semaphore can support (i.e. when the count associated with the *'Semaphore object'* becomes zero).

Counting Semaphore (continued)

- A real world example for the counting semaphore concept is the dormitory system for accommodation, as shown in the figure.
- 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 share 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.

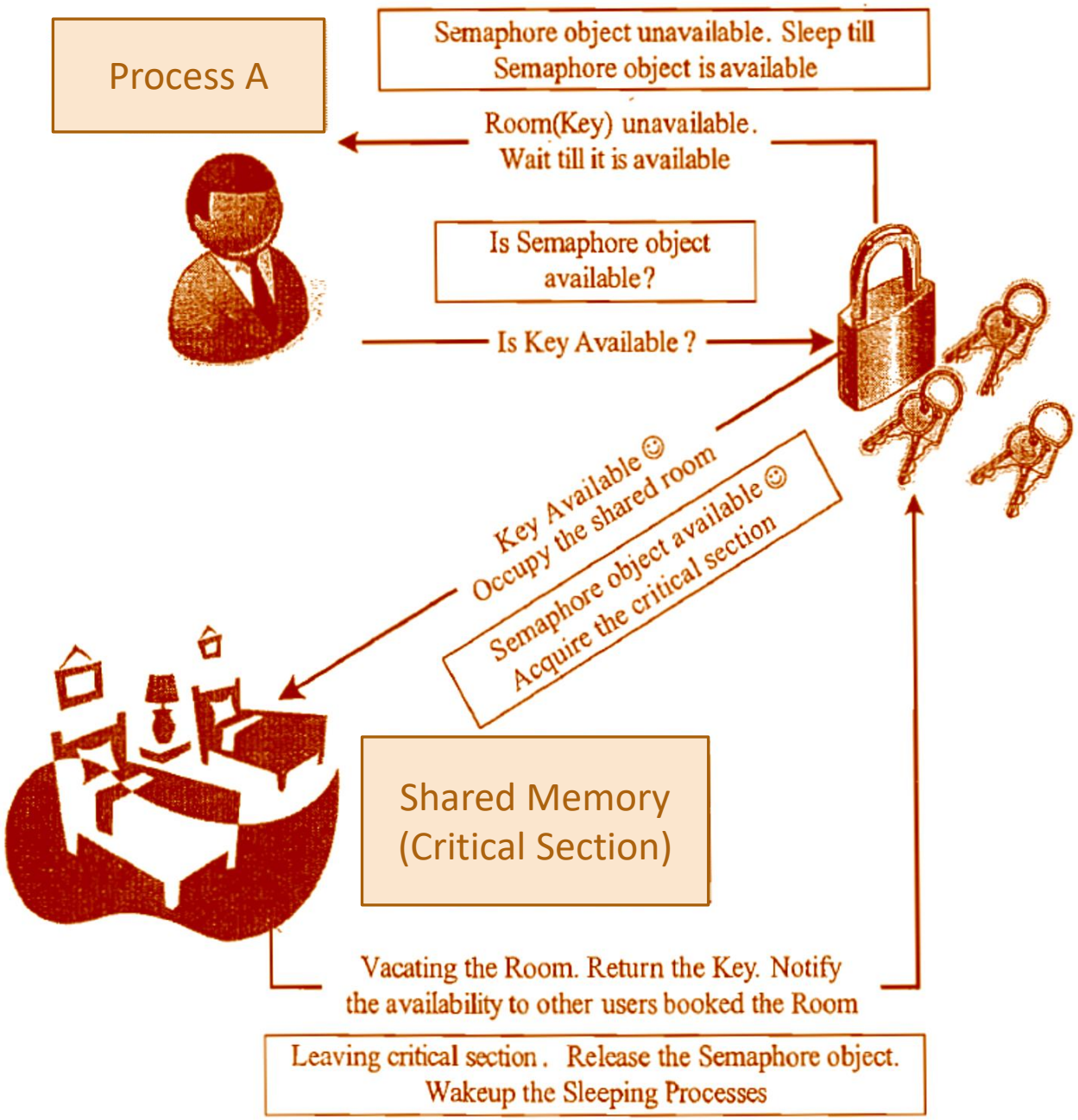


Fig.: The Concept of Counting Semaphore

Counting Semaphore vs. Binary Semaphore

- *Counting Semaphores* are similar to *Binary Semaphores* in operation.
- The only difference between *Counting Semaphore* and *Binary Semaphore* is that
  - Binary Semaphore can only be used for exclusive access, *whereas*
  - Counting Semaphores can be used for both
    - exclusive access (by restricting the maximum count value associated with the semaphore object to one at the time of creation of the semaphore object) *and*
    - limited access (by restricting the maximum count Value associated with the semaphore object to the limited number at the time of creation of the semaphore object)

Binary Semaphore (Mutex)

- *Binary Semaphore* (Mutex) is a synchronisation object provided by OS for process/thread synchronisation.
- 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.
- The state of a mutex object is set to ‘signalled’ when it is not owned by any process/thread, and set to ‘non-signalled’ when it is owned by any process/thread.

Binary Semaphore (Mutex) (continued)

- A real world example for the mutex concept is the hotel accommodation system (lodging system), as shown in the figure.
- 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.
  - 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.

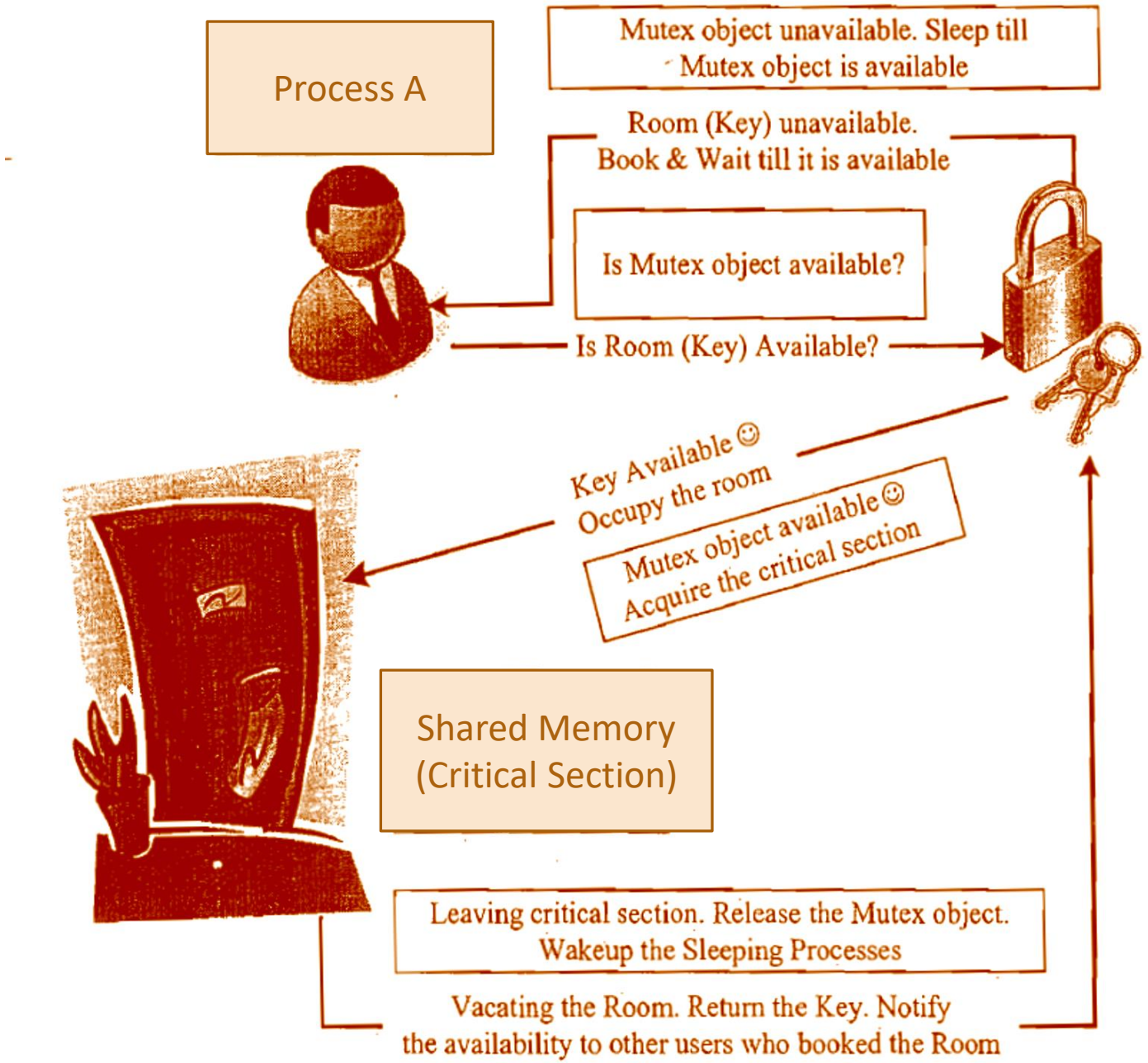


Fig.: The Concept of Binary Semaphore (Mutex)

How to Choose an RTOS

How to Choose an RTOS

- The decision of choosing an RTOS for an embedded design is very crucial.
- A lot of factors needs to be analysed carefully before making a decision on the selection of an RTOS.
- The requirements that needs to be analysed in the selection of an RTOS for an embedded design fall under two categories:
  - Functional requirements
  - Non-functional requirements

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.

Functional Requirements (continued)

- **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.

Functional Requirements (continued)

- **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.
- **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 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.

Functional Requirements (continued)

- **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.

Non-Functional Requirements (continued)

- **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.

Non-Functional Requirements (continued)

- **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.

Integration and Testing of Embedded Hardware and Firmware

Integration and Testing of Embedded Hardware and Firmware – Introduction

- Integration and testing of the embedded hardware and firmware is the immediate step following the embedded hardware and firmware development.
- Embedded hardware and firmware are developed in various steps.
  - The final embedded hardware constitute of a PCB with all necessary components affixed to it as per the original schematic diagram.
  - Embedded firmware represents the control algorithm and configuration data necessary to implement the product requirements on the product.
- The target embedded hardware without embedding the firmware is a dumb device and cannot function properly.
  - If you power up the hardware without embedding the firmware, the device may behave in an unpredicted manner.

Integration and Testing of Embedded Hardware and Firmware – Introduction (continued)

- Both embedded hardware and firmware should be independently tested (*Unit Tested*) to ensure their proper functioning.
- Functioning of individual hardware sections can be done by writing small utilities which checks the operation of the specified part.
- As far as the embedded firmware is concerned, its targeted functionalities can easily be checked by the simulator environment provided by the embedded firmware development tool's IDE (Integrated Development Environment).

Integration of Hardware and Firmware

- Integration of hardware and firmware deals with the embedding of firmware into the target hardware board.
  - It is the process of *'Embedding Intelligence'* to the product.
- For non-operating system based embedded products, if the processor/controller contains internal memory and the total size of the firmware is fitting into the code memory area, the code memory is downloaded into the target controller/processor.
- If the processor/controller does not support built in code memory or the size of the firmware is exceeding the memory size supported by the target processor/controller, an external dedicated EPROM/FLASH memory chip is used for holding the firmware.
  - This chip is interfaced to the processor/controller.
- A variety of techniques are used for embedding the firmware into the target board.

Out-of-Circuit Programming

- Out-of-circuit programming is performed outside the target board.
- The processor or memory chip into which the firmware needs to be embedded is taken out of the target board and it is programmed with the help of a *programming device* (also called *programmer*).

Fig.: Firmware Embedding Tool – Device Programmer: LabTool-48UXP

- The programming device is a dedicated unit which contains the necessary hardware circuit to generate the programming signals.

Out-of-Circuit Programming (continued)

- The programmer contains a ZIF socket with locking pin to hold the device to be programmed.
- The programming device will be under the control of a utility program running on a PC.
- Usually the programmer is interfaced to the PC through RS-232C/USB/Parallel Port Interface.
- The commands to control the programmer are sent from the utility program to the programmer through the interface.

Fig.: Universal Programmer

Out-of-Circuit Programming (continued)

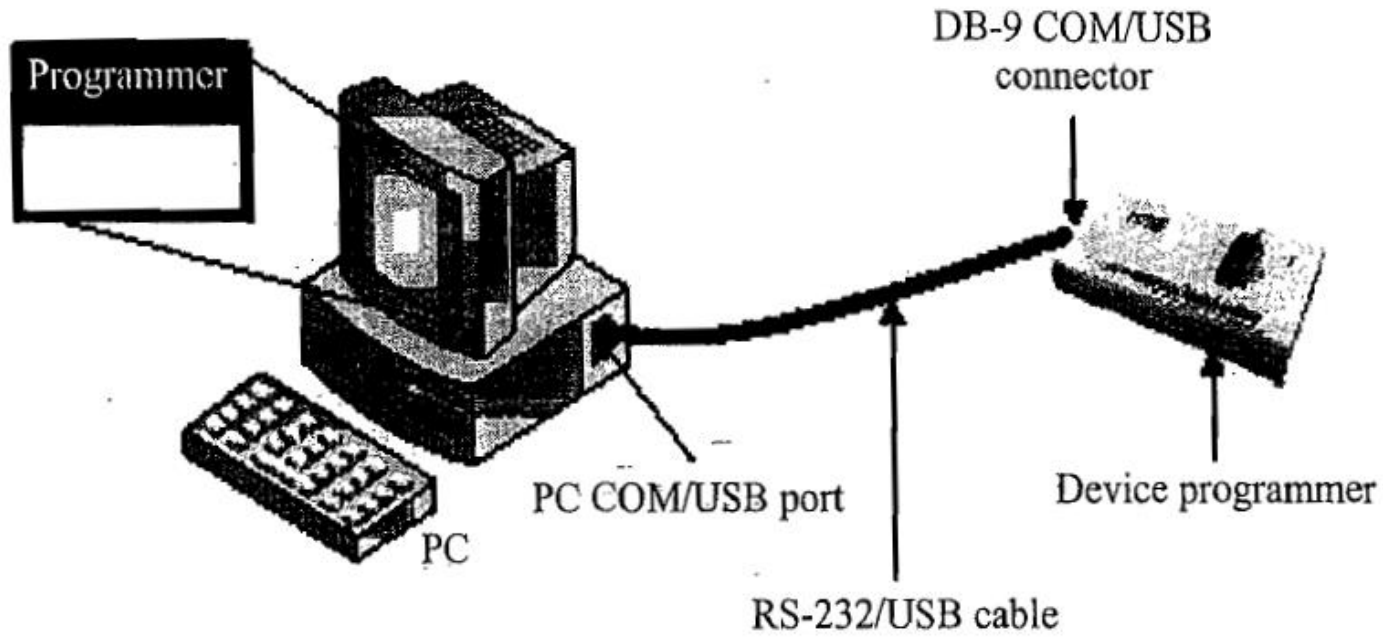


Fig.: Interfacing of Device Programmer with PC

Out-of-Circuit Programming (continued)

- The sequence of operations for embedding the firmware with a programmer is listed below:
  1. Connect the programming device to the specified port of PC (USB/COM port/parallel port).
  2. Power up the device (Ensure that the power indication LED is ON).
  3. Execute the programming utility on the PC and ensure proper connectivity is established between PC and programmer. In case of error, turn off device power and try connecting it again.
  4. Unlock the ZIF socket by turning the lock pin.
  5. Insert the device to be programmed into the open socket.
  6. Lock the ZIF socket.

Out-of-Circuit Programming (continued)

7. Select the device name from the list of supported devices.
8. Load the hex file which is to be embedded into the device.
9. Program the device by 'Program' option of utility program.
10. Wait till the completion of programming operation (Till busy LED of programmer is OFF).
11. Ensure that programming is successful by checking the status LED on the programmer (Usually 'Green' for success and 'Red' for error condition) or by noticing the feedback from the utility program.
12. Unlock the ZIF socket and take the device out of programmer.

Out-of-Circuit Programming (continued)

- Once the firmware is successfully embedded into the device, insert the device into the board, power up the board and test it for the required functionalities.
- If you want the firmware to be protected against unwanted external access, and if the device is supporting memory protection, enable the memory protection on the utility before programming the device.
- The programmer usually erases the existing content of the chip before programming the chip.
  - Only EEPROM and FLASH memory chips are erasable by the programmer.
  - Some old embedded systems may be built around UVEPROM chips and such chips should be erased using a separate 'UV Chip Eraser' before programming.

Out-of-Circuit Programming (continued)

- Drawbacks
  - The major drawback of out-of-circuit programming is the high development time.
    - Whenever the firmware is changed, the chip should be taken out of the development board for re-programming.
      - This is tedious and prone to chip damages due to frequent insertion and removal.
    - The programmer facilitates programming of only one chip at a time and it is not suitable for batch production.
      - Can be resolved using a 'Gang Programmer', which contains multiple ZIF sockets (4 to 8) and capable of programming multiple devices at a time.
      - But it is bit expensive compared to an ordinary programmer.
  - Another big drawback of out-of-circuit programming is that once the product is deployed in the market in a production environment, it is very difficult to upgrade the firmware.

Out-of-Circuit Programming (continued)

Fig.: Gang Programmer

Out-of-Circuit Programming (continued)

- Applications
  - The out-of-system programming technique is used for firmware integration for low end embedded products which runs without an operating system.
  - Out-of-circuit programming is commonly used for development of low volume products and Proof of Concept (PoC) product Development.

In System Programming (ISP)

- Here, the programming is done *'within the system',* meaning the firmware is embedded into the target device without removing it from the target board.
- It is the most flexible and easy way of firmware embedding.
  - The only pre-requisite is that the target device must have an ISP support.
- Apart from the target board, PC, ISP cable and ISP utility, no other additional hardware is required for ISP.
- The target board can be interfaced to the utility program running on PC through Serial Port/Parallel Port/USB.
- The communication between the target device and ISP utility will be in a serial format.
  - The serial protocols used for ISP may be 'Joint Test Action Group (JTAG)' or 'Serial Peripheral Interface (SPI)' or any other proprietary protocol.

In System Programming (ISP) (continued)

- In order to perform ISP operations, the target device should be powered up in a special ‘ISP mode’.
- ISP mode allows the device to communicate with an external host, such as a PC or terminal, through a serial interface.
- The device receives commands and data from the host, erases and reprograms code memory according to the received command.
- Once the ISP operations are completed, the device is re-configured so that it will operate normally by applying a reset or a re-power up.

In System Programming (ISP) (continued)

- Devices with *SPI - In System Programming* support contains a built-in SPI interface *(Serial Peripheral Interface)* and the on-chip EEPROM or FLASH memory is programmed through this interface.
- The primary I/O lines involved in SPI - In System Programming are:
  - MOSI - Master Out Slave In
  - MISO - Master In Slave Out
  - SCK - System Clock
  - RST - Reset of Target Device
  - GND - Ground of Target Device

In System Programming (ISP) (continued)

- PC acts as the master and target device acts as the slave in ISP.
- The program data is sent to the MOSI pin of target device and the device acknowledgement is originated from the MISO pin of the device.
- SCK pin acts as the clock for data transfer.
- Since the target device works under a supply voltage less than 5V (TTL/CMOS), it is better to connect these lines of the target device with the parallel port of the PC.
  - Since parallel port operations are also at 5V logic, no need for any other intermediate hardware for signal conversion.
- Standard SPI-ISP utilities are feely available on the internet and there is no need for going for writing own program.

In System Programming (ISP) (continued)

- For ISP operations, target device needs to be powered up in a pre-defined sequence.
- The power up sequence for In System Programming for Atmel's AT89S series microcontroller family is listed below:
  1. Apply supply voltage between VCC and GND pins of target chip.
  2. Set RST pin to "HIGH" state.
  3. If a crystal is not connected across pins XTAL1 and XTAL2, apply a 3 MHz to 24 MHz clock to XTAL1 pin and wait for at least 10 milliseconds.
  4. Enable serial programming by sending the Programming Enable serial instruction to pin MOSI/P1.5. The frequency of the shift clock supplied at pin SCK/P1.7 needs to be less than the CPU clock at XTAL1 divided by 40.
  5. The Code or Data array is programmed one byte at a time by supplying the address and data together with the appropriate Write instruction. The selected memory location is first erased before the new data is written. The write cycle is self-timed and typically takes less than 2.5 ms at 5V.
  6. Any memory location can be verified by using the Read instruction, which returns the content at the selected address at serial output MISO/P1 .6.
  7. After successfully programming the device, set RST pin low or turn off the chip power supply and turn it ON to commence the normal operation.

In System Programming (ISP) (continued)

- The key player behind ISP is a factory programmed memory (ROM) called *'Boot ROM'*.
- The *Boot ROM* normally resides at the top end of code memory space and it varies in the order of a few Kilo Bytes.
  - It contains a set of Low-level Instruction APIs and these APIs allow the processor/controller to perform the FLASH memory programming, erasing and reading operations.
- By default the Reset vector starts the code memory execution at location 0000H.
- If the ISP mode is enabled through the special ISP Power up sequence, the execution will start at the *Boot ROM* vector location.

In System Programming (ISP) (continued)

- In System Programming technique is the best advised programming technique for development work since the effort required to re-program the device in case of firmware modification is very little.
- Firmware upgrades for products supporting ISP is quite simple.

In Application Programming

- In Application Programming (IAP) is a technique used by the firmware running on the target device for modifying a selected portion of the code memory.
- It is not a technique for first time embedding of user written firmware.
- It modifies the program code memory under the control of the embedded application.
- Updating calibration data, look-up tables, etc., which are stored in code memory, are typical examples of IAP.

In Application Programming (continued)

- The *Boot ROM* resident API instructions which perform various functions such as programming, erasing, and reading the Flash memory during ISP-mode, are made available to the end-user written firmware for IAP.
  - Thus, it is possible for an end-user application to perform operations on the Flash memory.
- A common entry point to these API routines is provided for interfacing them to the end-user's application.
- Functions are performed by setting up specific registers as required by a specific operation and performing a call to the common entry point.
  - Like any other subroutine call, after completion of the function, control will return to the end-user's code.

In Application Programming (continued)

- The *Boot ROM* is shadowed with the user code memory in its address range.
- This shadowing is controlled by a status bit.
  - When this status bit is set, accesses to the internal code memory in this address range will be from the *Boot ROM*.
  - When cleared, accesses will be from the user's code memory.
- Hence the user should set the status bit prior to calling the common entry point for IAP operations.

Use of Factory Programmed Chip

- It is possible to embed the firmware into the target processor/controller memory at the time of chip fabrication itself.
  - Such chips are known as 'Factory programmed chips’.
- Once the firmware design is over and the firmware achieved operational stability, the firmware files can be sent to the chip fabricator to embed it into the code memory.
- Factory programmed chips are convenient for mass production applications and it greatly reduces the product development time.
- It is not recommended to use factory programmed chips for development purpose where the firmware undergoes frequent changes.
- Factory programmed ICs are bit expensive.

Firmware Loading for Operating System Based Devices

- The OS based embedded systems are programmed using the In System Programming (ISP) technique.
- OS based embedded systems contain a special piece of code called *'Boot loader'* program which takes control of the OS and application firmware embedding and copying of the OS image to the RAM of the system for execution.
- The *boot loader* for such embedded systems comes as pre-loaded or it can be loaded to the memory using the various interface supported like JTAG.
- The *boot loader* contains necessary driver initialisation implementation for initialising the supported interfaces like UART, TCP/IP etc.

Firmware Loading for Operating System Based Devices (continued)

- Boot loader implements menu options for selecting the source for OS image to load.
  - E.g. Load from FLASH ROM, Load from Network, Load through UART etc.
- In case of the network based loading, the boot loader broadcasts the target's presence over the network and the host machine on which the OS image resides can identify the target device by capturing this message.
  - Once a communication link is established between the host and target machine, the OS image can be directly downloaded to the FLASH memory of the target device.

Embedded System Development Environment

Embedded System Development Environment – Block Diagram

- The embedded system development environment consists of:
  - A Development Computer (PC) or Host, which acts as the heart of the development environment,
  - Integrated Development Environment (IDE) Tool for embedded firmware development and debugging,
  - Electronic Design Automation (EDA) Tool for Embedded Hardware design,
  - An emulator hardware for debugging the target board,
  - Signal sources (like Function generator) for simulating the inputs to the target board,
  - Target hardware debugging tools (Digital CRO, Multimeter, Logic Analyser, etc.) *and*
  - The target hardware.

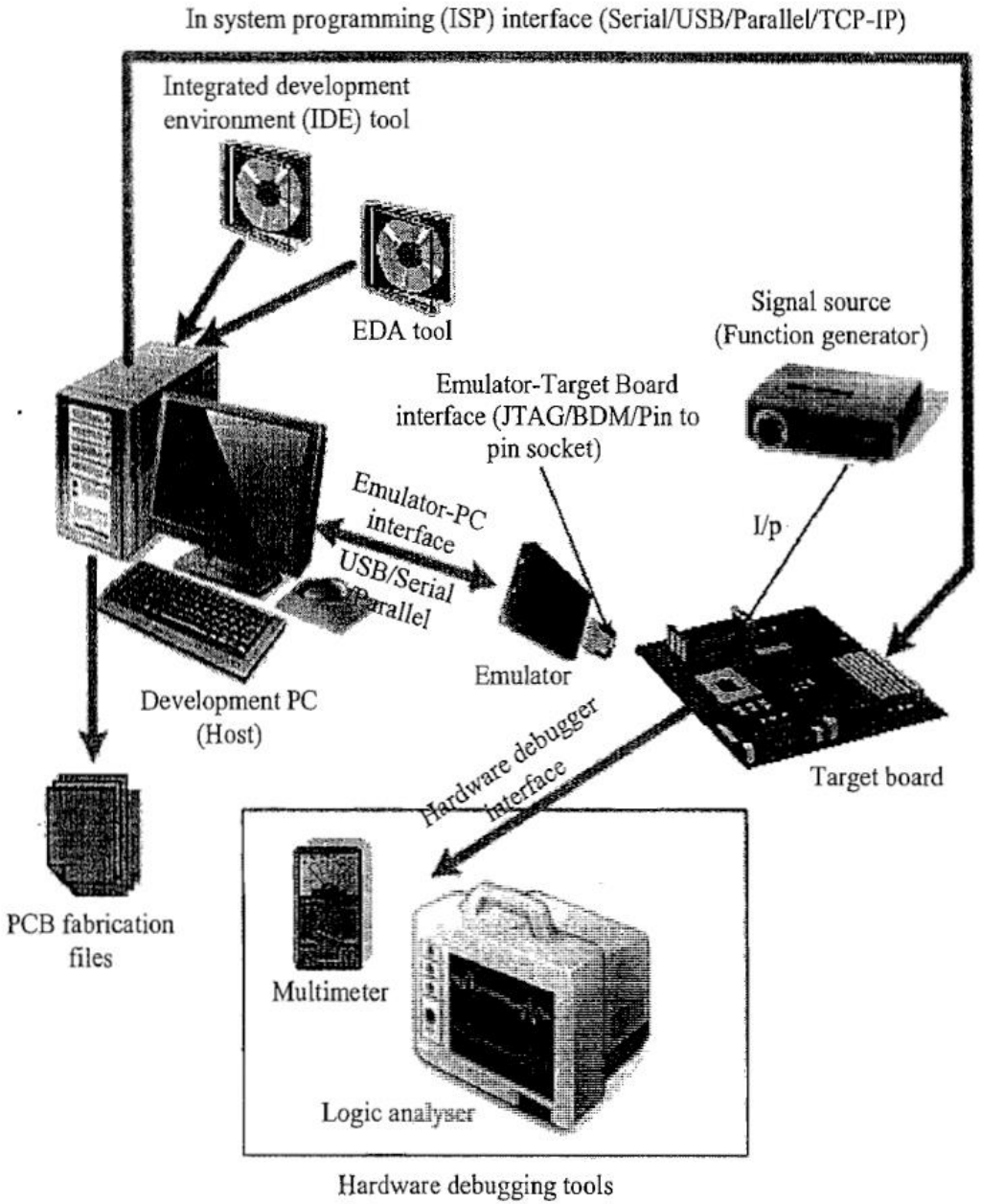


Fig.: The Embedded System Development Environment

Embedded System Development Environment – Block Diagram (continued)

- The Integrated Development Environment (IDE) and Electronic Design Automation (EDA) tools are selected based on the target hardware development requirement and they are supplied as Installable files in CDs by vendors.
- These tools need to be installed on the host PC used for development activities.
- These tools can be either freeware or licensed copy or evaluation versions.
  - Licensed versions of the tools are fully featured and fully functional whereas trial versions fall into two categories, tools with limited features, and full featured copies with limited period of usage.

Integrated Development Environment (IDE)

- 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 platform development),
  - Linker *and*
  - Debugger.

Integrated Development Environment (IDE) (continued)

- IDEs used in embedded firmware development are slightly different from the generic IDs 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.
  - Keil µVision from Keil software is an example for a third party IDE, which is used for developing embedded firmware for 8051 family microcontrollers and also ARM microcontrollers.
  - MPLAB is an IDE tool supplied by microchip for developing embedded firmware using their PIC family of microcontrollers.
  - CodeWarrior by Metrowerks is an example of IDE for ARM family of processors.

Disassembler/Decompiler

- *Disassembler* is a utility program which converts machine codes into target processor specific Assembly codes/instructions.
- The process of converting machine codes into Assembly code is known as 'Disassembling'.
  - In operation, disassembling is complementary to assembling/cross-assembling.
- *Decompiler* is the utility program for translating machine codes into corresponding high level language instructions.
- Decompiler performs the reverse operation of compiler/cross-compiler.
- The disassemblers/decompilers for different family of processors/controllers are different.

Disassembler/Decompiler (continued)

- Disassemblers/Decompilers are deployed in reverse engineering.
- *Reverse engineering* is the process of revealing the technology behind the working of a product.
- Reverse engineering in Embedded Product development is employed to find out the secret behind the working of popular proprietary products.
- Disassemblers/Decompilers help the reverse engineering process by translating the embedded firmware into Assembly/high level language instructions.
- Disassemblers/Decompilers are powerful tools for analysing the presence of malicious codes (virus information) in an executable image.

Simulators

- *Simulator* is a software tool used for simulating the various conditions for checking the functionality of the application firmware.
- The Integrated Development Environment (IDE) itself will be providing simulator support and they help in debugging the firmware for checking its required functionality.
- Simulators simulate the target hardware and the firmware execution can be inspected using simulators.
- The features of simulator based debugging are:
  - Purely software based
  - Doesn't require a real target system
  - Very primitive (Lack of featured I/O support. Everything is a simulated one)
  - Lack of Real-time behaviour

Simulators (continued)

- **Advantages of Simulator Based Debugging**
  - Simulator based debugging techniques are simple and straightforward.
- The major advantages of simulator based firmware debugging techniques are:
  - **No Need for Original Target Board**
    - Simulator based debugging technique is purely software oriented.
    - IDE's software support simulates the CPU of the target board.
    - User only needs to know about the memory map of various devices within the target board and the firmware should be written on the basis of it.
    - Since the real hardware is not required, firmware development can start well in advance immediately after the device interface and memory maps are finalised.
    - This saves development time.

Simulators (continued)

- **Simulate I/O Peripherals**
  - Simulator provides the option to simulate various I/O peripherals.
  - Using simulator's I/O support, the values for I/O registers can be edited and can be used as the input/output value in the firmware execution.
  - Hence it eliminates the need for connecting I/O devices for debugging the firmware.
- **Simulates Abnormal Conditions**
  - With simulator's simulation support, you can input any desired value for any parameter during debugging the firmware and can observe the control flow of firmware.
  - It really helps the developer in simulating abnormal operational environment for firmware and helps the firmware developer to study the behaviour of the firmware under abnormal input conditions.

Simulators (continued)

- **Limitations of Simulator Based Debugging**
  - **Deviation from Real Behaviour**
    - Simulation-based firmware debugging is always carried out in a development environment where the developer may not be able to debug the firmware under all possible combinations of input.
    - Under certain operating conditions we may get some particular result and it need not be the same when the firmware runs in a production environment.
  - **Lack of real-timeliness**
    - The major limitation of simulator based debugging is that it is not real-time in behaviour.
      - The debugging is developer driven and it is no way capable of creating a real-time behaviour.
    - Moreover in a real application the I/O condition may be varying or unpredictable.
      - Simulation goes for simulating those conditions for known values.

Emulators

- *Emulator* is hardware device which emulates the functionalities of the target device and allows real time debugging of the embedded firmware in a hardware environment.
- A circuit for emulating target device remains independent of a particular target system and processor.
- The emulator emulates the target system with extended memory and with code downloading ability during the edit-test-debug cycles.
- Emulators maintain the original look, feel, and behaviour of the embedded system.
- Even though the cost of developing an emulator is high, it proves to be the more cost efficient solution over time.
- Emulators allow software exclusive to one system to be used on another.
- It is more difficult to design emulators and it also requires better hardware than the original system.

Simulator vs. Emulator

- Simulator is a software application that precisely duplicates (mimics) the target CPU and simulates the various features and instructions supported by the target CPU.
- The simulator is a host-based program that imitates the functionality and instruction set of the target processor.
- In summary, the simulator *'simulates'* the target board CPU.

- 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.
- In summary, the emulator *'emulates'* the target board CPU.

Debuggers

- *Debugger* is a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables.
- *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 running as per the design.

Firmware Debugging

- Firmware debugging is performed to figure out the bug or the error in the firmware which creates the unexpected behaviour.
- There are several techniques for firmware debugging:
  - Incremental EEPROM Burning Technique
  - Inline Breakpoint Based Firmware Debugging
  - Monitor Program Based Firmware Debugging
  - In Circuit Emulator (ICE) Based Firmware Debugging
  - On Chip Firmware Debugging (OCD)

Incremental EEPROM Burning Technique

- This is the most primitive type of firmware debugging technique.
- In this technique, the code is separated into different functional code units.
- Instead of burning the entire code into the EEPROM chip at once, the code is burned in incremental order.
  - This means the code corresponding to all functionalities are separately coded, cross-compiled and burned into the chip one by one.
- In this technique, we are not doing any debugging, but we are observing the status of firmware execution as a debug method.
- Incremental firmware burning technique is widely adopted in small, simple system developments and in product development where time is not a big constraint (e.g. R&D projects).
  - It is also very useful in product development environments where no other debug tools are available.

Inline Breakpoint Based Firmware Debugging

- This is another primitive method of firmware debugging.
- Within the firmware where you want to ensure that firmware execution is reaching up to a specified point, an inline debug code is inserted immediately after the point.
- The debug code is a *printf()* function which prints a string given as per the firmware.
- The debug codes (*printf()* commands) can be inserted at each point where you want to ensure the firmware execution is covering that point.
- The source code is cross-compiled along with the debug codes embedded within it.
- The corresponding hex file is burned into the EEPROM.
- The *printf()* generated data can be viewed on the *HyperTerminal.*

Monitor Program Based Firmware Debugging

- This is the first adopted invasive method for firmware debugging.
- In this approach, a monitor program which acts as a supervisor is developed.
- The monitor program controls the downloading of user code into the code memory, inspects and modifies register/memory locations, allows single stepping of source code, etc.
- The monitor program implements the debug functions as per a pre-defined command set from the debug application interface.
- The first step in any monitor program development is determining a set of commands for performing various operations like firmware downloading, memory/register inspection/modification, single stepping, etc.

Monitor Program Based Firmware Debugging (continued)

- Once the commands for each operation is fixed, the code is written for performing the actions corresponding to these commands.
- The commands may be received through any of the external interface of the target processor (e.g. RS-232C serial interface/parallel interface/USB, etc.).
  - The monitor program should query this interface to get commands or should handle the command reception if the data reception is implemented through interrupts.
- On receiving a command, it is examined and the action corresponding to it is performed.
- The entire code stuff handling the command reception and corresponding action implementation is known as the "*Monitor Program*".
- After the successful completion of the *'Monitor Program'* development, it is compiled and burned into the FLASH memory or ROM of the target board.
- The code memory containing the monitor program is known as the *'Monitor ROM'*.

Monitor Program Based Firmware Debugging (continued)

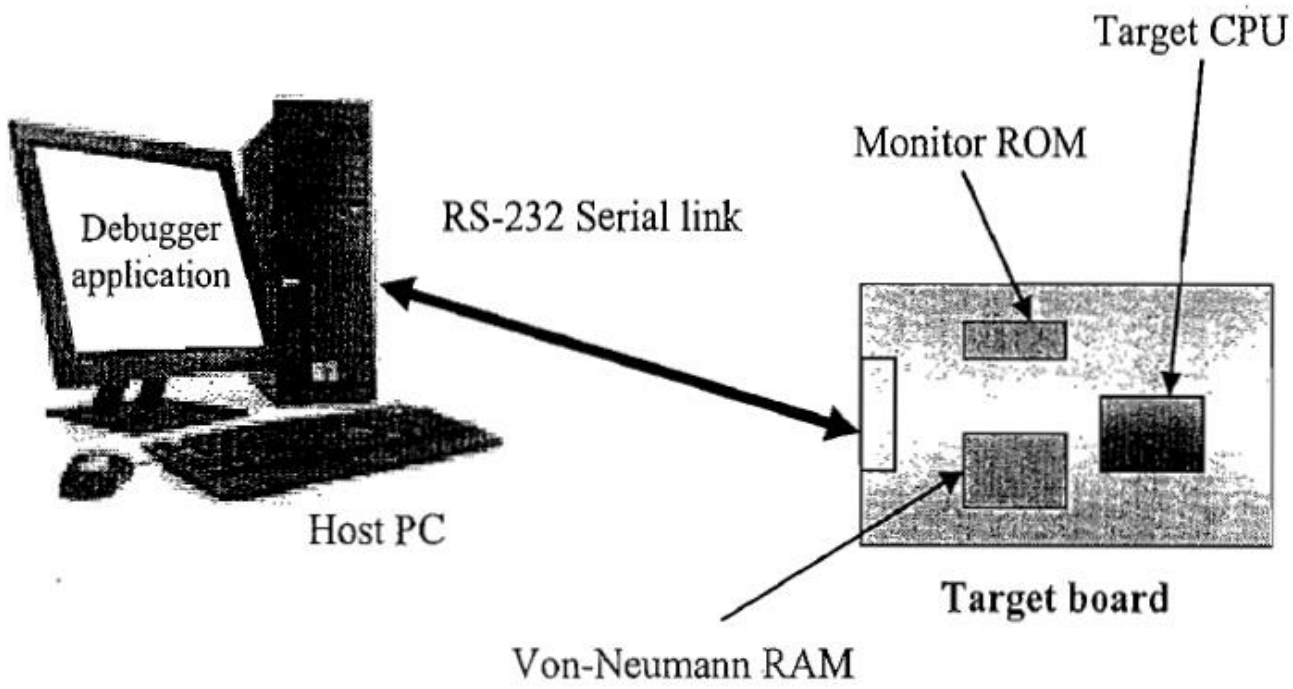


Fig.: Monitor Program Based Target Firmware Debug Setup

In Circuit Emulator (ICE) Based Firmware Debugging

- Emulator is a special hardware device used for emulating the functionality of a processor/controller and performing various debug operations like halt firmware execution, set breakpoints, get or set internal RAM/CPU register, etc.
- Nowadays pure software applications which perform the functioning of a hardware emulator is also called as 'Emulators' (though they are 'Simulators' in operation).
- The emulator application for emulating the operation of a PDA phone for application development is an example of a 'Software Emulator’.
- A hardware emulator is controlled by a debugger application running on the development PC.
  - Most of the IDEs incorporate debugger support for some of the emulators commonly available in the market.

In Circuit Emulator (ICE) Based Firmware Debugging (continued)

- Figure illustrates the different subsystems and interfaces of an 'Emulator' device.

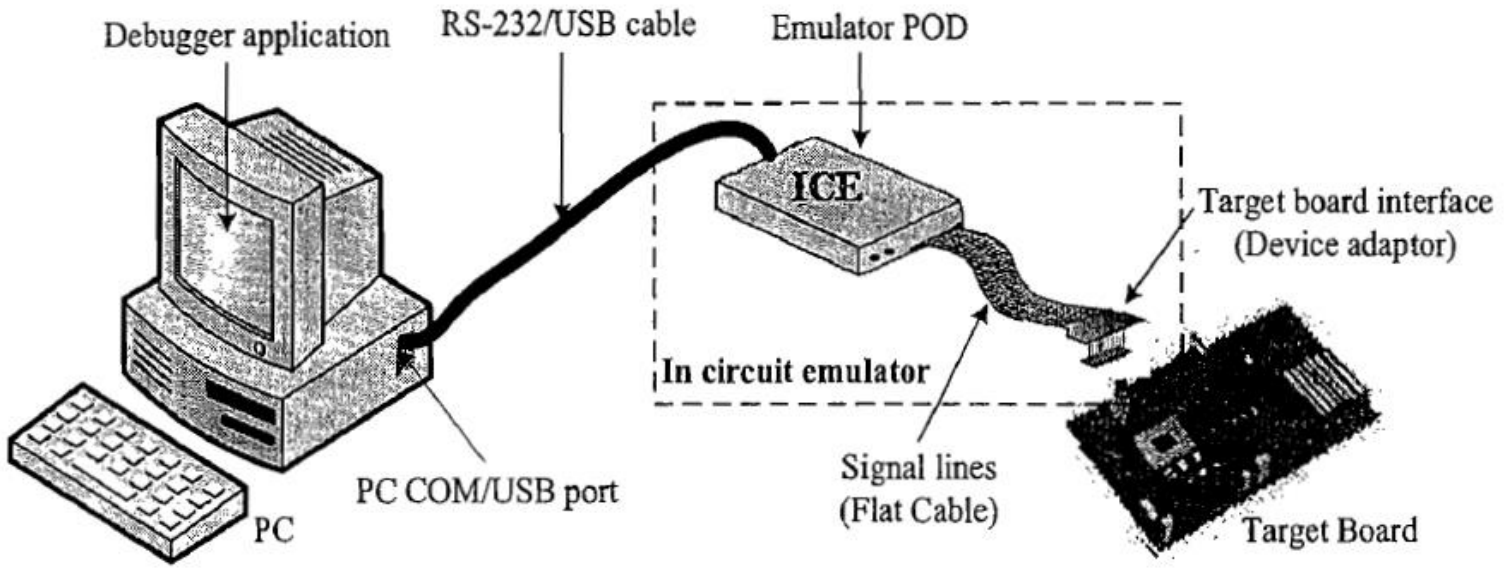


Fig.: In Circuit Emulator (ICE) Based Target Debugging

On Chip Firmware Debugging (OCD)

- Modern processors/controllers incorporate built in debug modules called On Chip Debug (OCD) support.
- Though OCD adds silicon complexity and cost factor, from a developer perspective it is a very good feature supporting fast and efficient firmware debugging.
- The On Chip Debug facilities integrated to the processor/controller are chip vendor dependent and most of them are proprietary technologies like Background Debug Mode (BDM), OnCE, etc.
- Some vendors add 'on chip software debug support' through JTAG (Joint Test Action Group) port.
- Usually the on-chip debugger provides the means to set simple breakpoints, query the internal state of the chip and single step through code.
- Background Debug Mode (BDM) and JTAG (Joint Test Action Group) are two commonly used interfaces for OCD.
- OCD module implements dedicated registers for controlling debugging.

Target Hardware Debugging

- Hardware debugging involves the monitoring of various signals of the target board (address/data lines, port pins, etc.), checking the interconnection among various components, circuit continuity checking, etc.
- The various hardware debugging tools used in embedded product development are:
  - Magnifying Glass (Lens)
  - Multimeter
  - Digital CRO
  - Logic Analyser
  - Function Generator

Magnifying Glass (Lens)

- Magnifying glass is the primary hardware debugging tool used for embedded hardware debugging.
- A magnifying glass is a powerful visual inspection tool.
- With a magnifying glass (lens), the surface of the target board can be examined thoroughly for dry soldering of components, missing components, improper placement of components, improper soldering, track (PCB connection) damage, short of tracks, etc.
- Nowadays high quality magnifying stations are available for visual inspection.
- The magnifying station incorporates magnifying glasses attached to a stand with CFL tubes for providing proper illumination for inspection.
- The station usually incorporates multiple magnifying lenses.
- The main lens acts as a visual inspection tool for the entire hardware board whereas the other small lens within the station is used for magnifying a relatively small area of the board which requires thorough inspection.

Multimeter

- A multimeter is used for measuring various electrical quantities like voltage (Both AC and DC), current (DC and AC), resistance, capacitance, continuity checking, transistor checking, cathode and anode identification of diode, etc.
- Any multimeter will work over a specific range for each measurement.
- A multimeter is the most valuable tool in the toolkit of an embedded hardware developer.
- It is the primary debugging tool for physical contact based hardware debugging.
- In embedded hardware debugging, it is mainly used for checking the circuit continuity between different points on the board, measuring the supply voltage, checking the signal value, polarity, etc.
- Both analog and digital versions of a multimeter are available.
  - The digital version is preferred over analog the one for various reasons like readability, accuracy, etc.

Digital CRO

- Cathode Ray Oscilloscope (CRO) is used for waveform capturing and analysis, measurement of signal strength, etc.
- CRO is a very good tool in analysing interference noise in the power supply line and other signal lines.
- Monitoring the crystal oscillator signal from the target board is a typical example of the usage of CRO for waveform capturing and analysis in target board debugging.
- CROs are available in both analog and digital versions.
  - Though Digital CROs are costly, featurewise they are best suited for target board debugging applications.
- Digital CROS are available for high frequency support and they also incorporate modern techniques for recording waveform over a period of time, capturing waves on the basis of a configurable event (trigger) from the target board.
- Most of the modern digital CROs contain more than one channel and it is easy to capture and analyse various signals from the target board using multiple channels simultaneously.
- Various measurements like phase, amplitude, etc. is also possible with CROs.

Logic Analyser

- Logic analyser is used for capturing digital data (logic 1 and 0) from a digital circuitry whereas CRO is employed in capturing all kinds of waves including logic signals.
- A logic analyser contains special connectors and clips which can be attached to the target board for capturing digital data.
- In target board debugging applications, a logic analyser captures the states of various port pins, address bus and data bus of the target processor/controller, etc.
- Logic analysers give an exact reflection of what happens when particular line of firmware is running.
- This is achieved by capturing the address line logic and data line logic of target hardware.
- Most modern logic analysers contain provisions for storing captured data, selecting a desired region of the captured waveform, zooming selected region of the captured waveform, etc.

Function Generator

- Function generator is not a debugging tool.
- It is an input signal simulator tool.
- A function generator is capable of producing various periodic waveforms like sine wave, square wave, saw-tooth wave, etc. with different frequencies and amplitude.
- Sometimes the target board may require some kind of periodic waveform with a particular frequency as input to some part of the board.
- Thus, in a debugging environment, the function generator serves the purpose of generating and supplying required signals.

Boundary Scan

- *Boundary scan* is a technique used for testing the interconnection among the various chips, which support JTAG interface, present in the board.
- Boundary scan is also widely used as a debugging method to watch integrated circuit pin states, measure voltage, or analyse sub-blocks inside an integrated circuit.
- The boundary scan test architecture provides a means to test interconnects between integrated circuits on a board without using physical test probes.
- It adds a boundary scan cell that includes a multiplexer and latches, to each pin on the device.

Boundary Scan (continued)

- *Boundary Scan Description Language (BSDL)* is used for implementing boundary scan tests using JTAG.
  - BSDL is a subset of VHDL and it describes the JTAG implementation in a device.
- The benefits provided by boundary scan are:
  - Lower test generation costs
  - Reduced test time
  - Reduced time to market
  - Simpler and less costly testers
  - Compatibility with tester interfaces
  - High-density packaging devices accommodation

References

1. Shibu K V, ***"Introduction to Embedded Systems"***, Tata McGraw Hill, 2009.
2. Raj Kamal, ***"Embedded Systems: Architecture and Programming",*** Tata McGraw Hill, 2008.