10.1 OPERATING SYSTEM BASICS

LO 1 Understand the basics of an operating system and the need for an operating system

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

- Make the system convenient to use
- Organise and manage the system resources efficiently and correctly

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

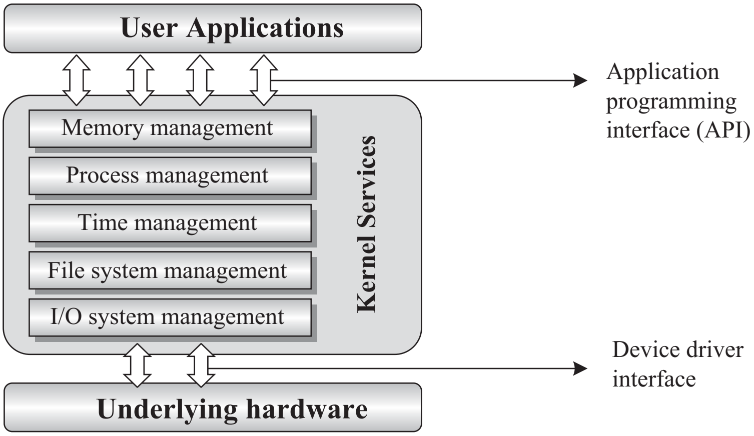


Fig. 10.1 The Operating System Architecture

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

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. We will look into the description of process and process management in a later section of this chapter.

Primary Memory Management The term primary memory refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored. The Memory Management Unit (MMU) of the kernel is responsible for

- Keeping track of which part of the memory area is currently used by which process
- Allocating and De-allocating memory space on a need basis (Dynamic memory allocation).

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

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

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. This list may be available in advance, at the time of building the kernel. Some kernels, dynamically updates the list of available devices as and when a new device is installed (e.g. Windows NT kernel keeps the list updated when a new plug '*n*' play USB device is attached to the system). The service 'Device Manager' (Name may vary across different OS kernels) of the kernel is responsible for handling all I/O device related operations. The kernel talks to the I/O device through a set of low-level systems calls, which are implemented in a service, called device drivers. The device drivers are specific to a device or a class of devices. The Device Manager is responsible for

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

Secondary Storage Management The secondary storage management deals with managing the secondary storage memory devices, if any, connected to the system. Secondary memory is used as backup medium for programs and data since the main memory is volatile. In most of the systems, the secondary storage is kept 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. Windows 10 with user permissions like 'Administrator', 'Standard', 'Restricted', etc.). Protection deals with implementing the security policies to restrict the access to both user and system resources by different applications or processes or users. In multiuser supported operating systems, one user may not be allowed to view or modify the whole/portions of another user's data or profile details. In addition, some application may not be granted with permission to make use of some of the system resources. This kind of protection is provided by the protection services running within the kernel.

Interrupt Handler Kernel provides handler mechanism for all external/internal interrupts generated by the system.

These are some of the important services offered by the kernel of an operating system. It does not mean that a kernel contains no more than components/services explained above. Depending on the type of the operating system, a kernel may contain lesser number of components/services or more number of components/services. In addition to the components/services listed above, many operating systems offer a number of add-on system components/services to the kernel. Network communication, network management, user-interface graphics, timer services (delays, timeouts, etc.), error handler, database management, etc. are examples for such components/services. Kernel exposes the interface to the various kernel applications/services, hosted by kernel, to the user applications through a set of standard Application Programming Interfaces (APIs). User applications can avail these API calls to access the various kernel application/services.

10.1.1.1 Kernel Space and User Space

As we discussed in the earlier section, the applications/services are classified into two categories, namely: user applications and kernel applications. The program code corresponding to the kernel applications/services are kept in a contiguous area (OS dependent) of primary (working) memory and is protected from the un-authorised access by user programs/applications. The memory space at which the kernel code is located is known as '*Kernel Space*'. Similarly, all user applications are loaded to a specific area of primary memory and this memory area is referred as '*User Space*'. User space is the memory area where user applications are loaded and executed. The partitioning of memory into kernel and user space is purely Operating System dependent. Some OS implements 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; Meaning, the entire code for the user application need not be loaded to the main (primary) memory at once; instead 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. Each process run in its own virtual memory space and are not allowed accessing the memory space corresponding to another processes, unless explicitly requested by the process. Each process will have certain privilege levels on accessing the memory of other processes and based on the privilege settings, processes can request kernel to map another process's memory to its own or share through some other mechanism. Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory.

10.1.1.2 Monolithic Kernel and Microkernel

As we know, 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*' and '*Micro*'.

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. The architecture representation of a monolithic kernel is given in Fig. 10.2.

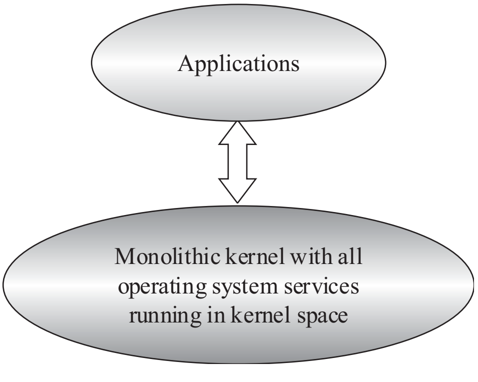


Fig. 10.2 The Monolithic Kernel Model

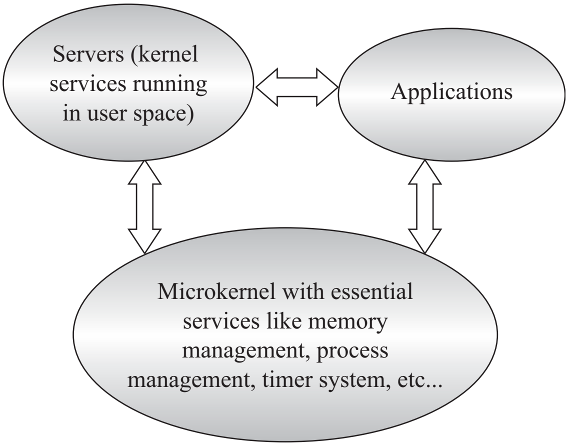


Fig. 10.3 The Microkernel model

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 abstraction 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. The architecture representation of a microkernel is shown in Fig. 10.3.

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*'. Refer Chapter 3 to get an understanding of '*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 services, which run as '*Server*' application can be changed without the need to restart the whole system. This makes the system dynamically configurable.

10.2 TYPES OF OPERATING SYSTEMS

LO 2 Classify the 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.

10.2.1 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 an OS 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 10/8.x/XP/MS-DOS etc are examples for General Purpose Operating Systems.

10.2.2 Real-Time Operating System (RTOS)

There is no universal definition available for the term '*Real-Time*' when it is used in conjunction with operating systems. What '*Real-Time*' means in Operating System context is still a debatable topic and there are many definitions available. In a broad sense, '*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 Embedded Compact, QNX, VxWorks MicroC/OS-II etc are examples of Real Time Operating Systems (RTOS).

10.2.2.1 The Real-Time Kernel

The kernel of a Real-Time Operating System is referred as Real. Time kernel. In complement to the conventional OS 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 listed below:

- Task/Process management
- Task/Process scheduling
- Task/Process synchronisation
- Error/Exception handling
- Memory management
- Interrupt handling
- Time management

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 and task/process termination/deletion. A Task Control Block (TCB) is used for holding the information corresponding to a task. 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: 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: 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 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

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 optimal scheduling of tasks to provide a deterministic behaviour. We will discuss the various types of scheduling in a later section of this chapter.

Task/Process Synchronisation Deals with synchronising the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks. We will discuss the various synchronisation techniques and inter task /process communication in a later section of this chapter.

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). *GetLastError()* API provided by Windows CE/Embedded Compact RTOS is an example for such a system call. Watchdog timer is a mechanism for handling the timeouts for tasks. Certain tasks may involve the waiting of external events from devices. These tasks will wait infinitely when the external device is not responding and the task will generate a hang-up behaviour. In order to avoid these types of scenarios, a proper timeout mechanism should be implemented. A watchdog is normally used in such situations. The watchdog will be loaded with the maximum expected wait time for the event and if the event is not triggered within this wait time, the same is informed to the task and the task is timed out. If the event happens before the timeout, the watchdog is resetted.

Memory Management Compared to the General Purpose Operating Systems, the memory management function of an RTOS kernel is slightly different. In general, the memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block (initialised memory block consumes more allocation time than un-initialised memory block). Since predictable timing and deterministic behaviour are the primary focus of an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation. 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 and the kernel enters a *fail-safe* mode when an illegal memory access occurs.

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 memory allocation can be implemented as constant functions and thereby it consumes fixed amount of time for memory allocation. This leaves the deterministic behaviour of the RTOS kernel untouched. The '*block*' memory concept avoids the garbage collection overhead also. (We will explore this technique under the MicroC/OS-II kernel in a latter chapter).The '*block*' based memory

---

[*] *Virtual Memory* is an imaginary memory supported by certain operating systems. Virtual memory expands the address space available to a task beyond the actual physical memory (RAM) supported by the system. Virtual memory is implemented with the help of a Memory Management Unit (MMU) and 'memory paging'. The program memory for a task can be viewed as different pages and the page corresponding to a piece of code that needs to be executed is loaded into the main physical memory (RAM). When a memory page is no longer required, it is moved out to secondary storage memory and another page which contains the code snippet to be executed is loaded into the main memory. This memory movement technique is known as demand paging. The MMU handles the demand paging and converts the virtual address of a location in a page to corresponding physical address in the RAM.

allocation achieves deterministic behaviour with the trade-of limited choice of memory chunk size and suboptimal memory usage.

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*. Interrupts which occurs in sync with the currently executing task is known as *Synchronous* interrupts. Usually the software interrupts fall under the Synchronous Interrupt 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. Asynchronous interrupts are interrupts, which occurs 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 (Depends on OS kernel implementation) 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 interrupts 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.

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*'. The '*Timer tick*' 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 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

$$2^{32} * 10^{-6} / (24 * 60 * 60) = 49700 \text{ Days} = \sim 0.0497 \text{ Days} = 1.19 \text{ Hours}$$

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

$$2^{32} * 10^{-3} / (24 * 60 * 60) = 497 \text{ Days} = 49.7 \text{ Days} = \sim 50 \text{ Days}$$

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.

Apart from these basic functions, some RTOS provide other functionalities also (Examples are file management and network functions). Some RTOS kernel provides options for selecting the required kernel functions at the time of building a kernel. The user can pick the required functions from the set of available functions and compile the same to generate the kernel binary. Windows CE is a typical example for such an RTOS. While building the target, the user can select the required components for the kernel.

10.2.2.2 Hard Real-Time

Real-Time Operating Systems that strictly adhere to the timing constraints for a task is referred as '*Hard Real-Time*' systems. A Hard Real-Time system must meet the deadlines for a task without any slippage. Missing any deadline may produce catastrophic results for Hard Real-Time Systems, including permanent data lose and irrecoverable damages to the system/users. Hard Real-Time systems emphasise the principle '*A late answer is a wrong answer*'. A system can have several such tasks and the key to their correct operation lies in scheduling them so that they meet their time constraints. Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples for Hard Real-Time Systems. The Air bag control system should be into action and deploy the air bags when the vehicle meets a severe accident. Ideally speaking, the time for triggering the air bag deployment task, when an accident is sensed by the Air bag control system, should be zero and the air bags should be deployed exactly within the time frame, which is predefined for the air bag deployment task. Any delay in the deployment of the air bags makes the life of the passengers under threat. When the air bag deployment task is triggered, the currently executing task must be pre-empted, the air bag deployment task should be brought into execution, and the necessary I/O systems should be made readily available for the air bag deployment task. To meet the strict deadline, the time between the air bag deployment event triggering and start of the air bag deployment task execution should be minimum, ideally zero. As a rule of thumb, 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. In general, the presence of *Human in the loop* (*HITL*) for tasks introduces unexpected delays in the task execution. Most of the Hard Real-Time Systems are automatic and does not contain a 'human in the loop'.

10.2.2.3 Soft Real-Time

Real-Time Operating System that does not guarantee meeting deadlines, but offer the best effort to meet the deadline are referred as '*Soft Real-Time*' systems. Missing deadlines for tasks are acceptable for a Soft Real-time system if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS). 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.

10.3 TASKS, PROCESS AND THREADS

LO 3 Discuss tasks, processes and threads in the operating system context

The term '*task*' refers to something that needs to be done. In our day-to-day life, we are bound to the execution of a number of tasks. The task can be the one assigned by our managers or the one assigned by our professors/teachers or the one related to our personal or family needs. In addition, we will have an order of priority and schedule/timeline for executing these tasks. 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.

10.3.1 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.

10.3.1.1 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 Fig. 10.4.

A process which inherits all the properties of the CPU can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor. When the process gets its turn, its registers and the program counter register becomes mapped to the physical registers of the CPU. From a memory perspective, the memory occupied by the *process* is segregated into three regions, namely, Stack memory, Data memory and Code memory (Fig. 10.5).

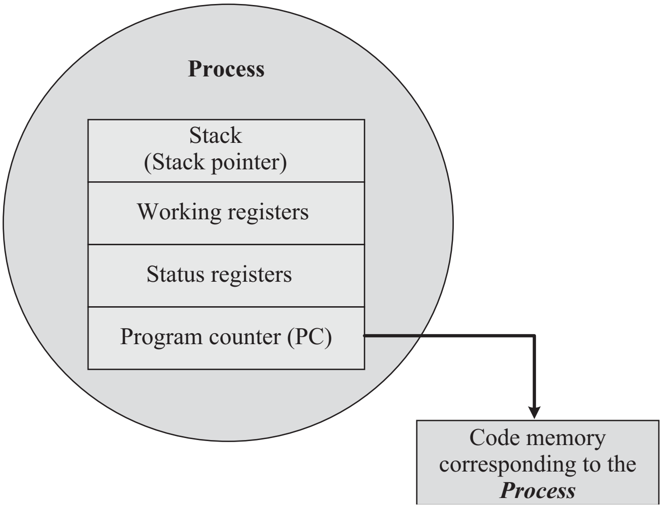


Fig. 10.4 Structure of a Process

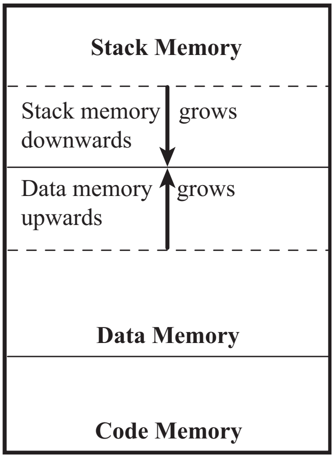


Fig. 10.5 Memory organisation of a Process

The 'Stack' memory holds all temporary data such as variables local to the process. Data memory holds all global data for the process. The code memory contains the program code (instructions) corresponding to the process. On loading a process into the main memory, a specific area of memory is allocated for the process. The stack memory usually starts (OS Kernel implementation dependent) at the highest memory address from the memory area allocated for the process. Say for example, the memory map of the memory area allocated for the process is 2048 to 2100, the stack memory starts at address 2100 and grows downwards to accommodate the variables local to the process.

10.3.1.2 Process States and State Transition

The creation of a process to its termination is not a single step operation. The process traverses through a series of states during its transition from the newly created state to the terminated state. The cycle through which a process changes its state from '*newly created*' to '*execution completed*' is known as '*Process Life Cycle*'. The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to do next. Figure 10.6 represents the various states associated with a process.

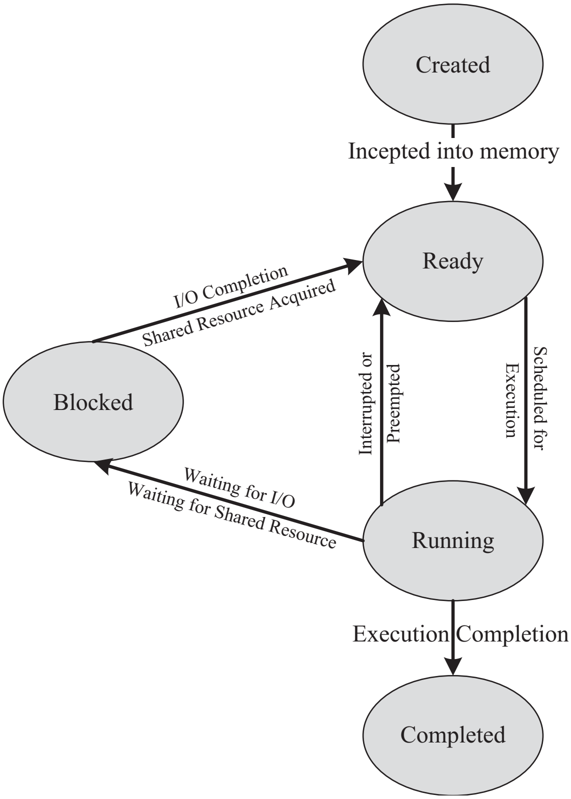


Fig. 10.6 Process states and state transition representation

The state at which a process is being created is referred as 'Created State'. The Operating System recognises a process in the '*Created State*' but no resources are allocated to the process. The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as '*Ready State*'. At this stage, the process is 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. '*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 (will be discussed at a later section of this chapter). A state where the process completes its execution is known as '*Completed State*'. The transition of a process from one state to another is known as '*State transition*'. When a process changes its state from Ready to running or from running to blocked or terminated or from blocked to running, the CPU allocation for the process may also change.

It should be noted that the state representation for a process/task mentioned here is a generic representation. The states associated with a task may be known with a different name or there may be more or less number of states than the one explained here under different OS kernel. For example, under VxWorks' kernel, the tasks may be in either one or a specific combination of the states READY, PEND, DELAY and SUSPEND. The PEND state represents a state where the task/process is blocked on waiting for I/O or system resource. The DELAY state represents a state in which the task/process is sleeping and the SUSPEND state represents a state where a task/process is temporarily suspended from execution and not available for execution. Under MicroC/OS-II kernel, the tasks may be in one of the states, DORMANT, READY, RUNNING, WAITING or INTERRUPTED. The DORMANT state represents the 'Created' state and WAITING state represents the state in which a process waits for shared resource or I/O access. We will discuss about the states and state transition for tasks under VxWorks and uC/OS-II kernel in a later chapter.

10.3.1.3 Process Management

Process management deals with the 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 and process termination/deletion. For more details on Process Management, refer to the section 'Task/Process management' given under the topic 'The Real-Time Kernel' of this chapter.

10.3.2 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 lightweight process. A process can have many threads of execution. Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area. Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack. The memory model for a process and its associated threads are given in Fig. 10.7.

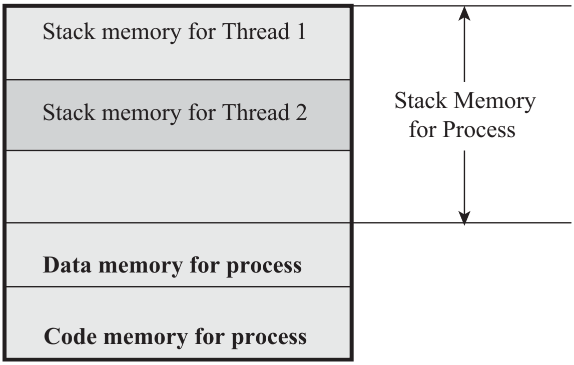


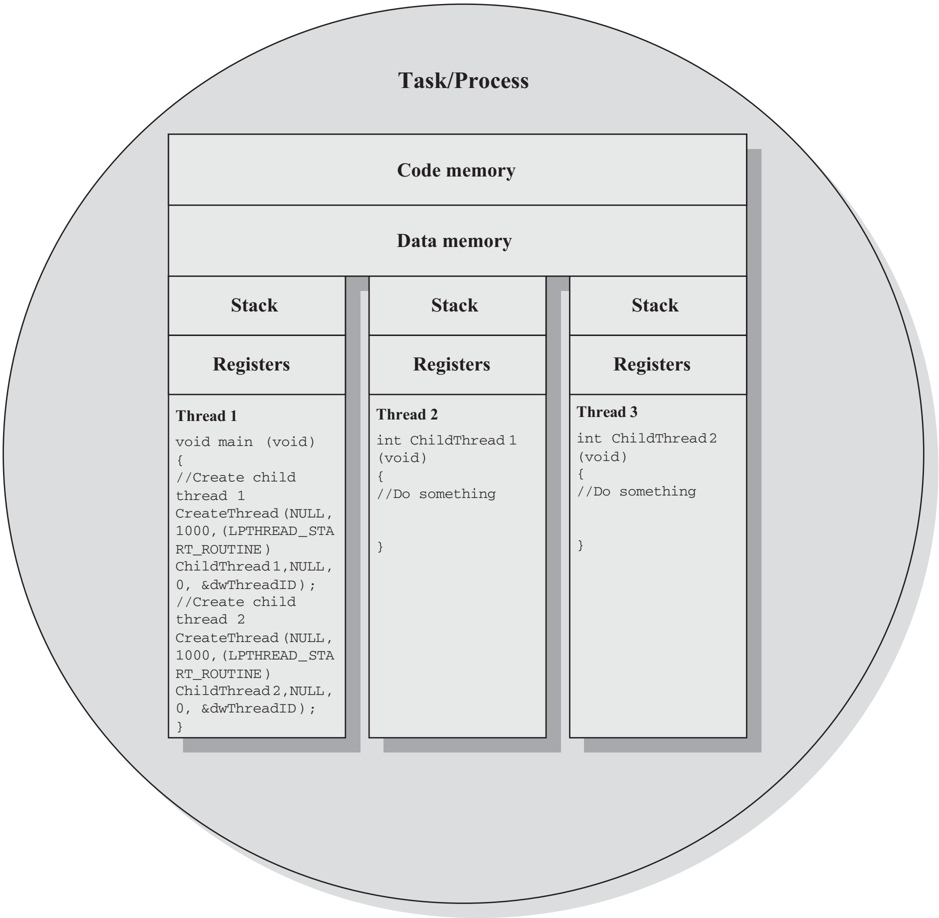
Fig. 10.7 Memory organisation of a Process **and its associated** Threads

10.3.2.1 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. Instead of this single sequential execution of the whole process, if the task/process is split into different threads carrying out the different subfunctionalities of the process, the CPU can be effectively utilised and when the thread corresponding to the I/O operation enters the wait state, another threads which do not require the I/O event for their operation can be switched into execution. This leads to more speedy execution of the process and the efficient utilisation of the processor time and resources. The multithreaded architecture of a process can be better visualised with the thread-process diagram shown in Fig. 10.8.

If the process is split into multiple threads, which executes a portion of the process, there will be a main thread and rest of the threads will be created within the main thread. Use of multiple threads to execute a process brings the following advantage.

- Better memory utilisation. Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.
- Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilised by other threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of the process.
- Efficient CPU utilisation. The CPU is engaged all time.



```
void main (void)
{
//Create child
thread 1
CreateThread(NULL,
1000,(LPTHREAD_STA
RT_ROUTINE)
ChildThread1,NULL,
0, &dwThreadID);
//Create child
thread 2
CreateThread(NULL,
1000,(LPTHREAD_STA
RT_ROUTINE)
ChildThread2,NULL,
0, &dwThreadID);
}
```

```
int ChildThread1
(void)
{
//Do something


}
```

```
int ChildThread2
(void)
{
//Do something


}
```

Fig. 10.8 Process **with multi-threads**

10.3.2.2 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 explained below.

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.

The primitive

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

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*' (It is a void * in the above example). 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).

The primitive

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

blocks the current thread and waits until the completion of the thread pointed by it (In this example *new_thread*)

All the POSIX 'thread calls' returns an integer. A return value of zero indicates the success of the call. It is always good to check the return value of each call.

Example 1

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" );
    //Wait for some time. Do nothing
    //The following line of code can be replaced with
    //OS supported delay function like sleep(), delay () etc…
    for( i= 0; i < 10000; i++ );
   }
  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" );
     //Wait for some time. Do nothing
     //The following line of code can be replaced with
     //OS supported delay function like sleep(), delay etc…
     for( i= 0; i < 10000; i++ );
    }
   if (pthread_join(tcb, NULL ))
     {
        //Thread join failed
        printf("Error in Thread join\n" );
        return -1;
     }
   return 1;
 }
```

You can compile this application using the *gcc* compiler. Examine the output to figure out the thread execution switching. The lines printed will give an idea of the order in which the thread execution is switched between. The *pthread_join* call forces the main thread to wait until the completion of the thread *tcb*, if the main thread finishes the execution first.

The termination of a thread can happen in different ways. The thread can terminate either by completing its execution (natural termination) or by a forced termination. In a natural termination, the thread completes its execution and returns back to the main thread through a simple *return* or by executing the *pthread_exit()* call. Forced termination 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.

pthread_exit() call is used by a thread to explicitly exit after it completes its work and is no longer required to exist. If the main thread finishes before the threads it has created, and exits with *pthread_exit()*, the other threads continue to execute. If the main thread uses *exit* call to exit the thread, all threads created by the main thread is terminated forcefully. Exiting a thread with the call *pthread_exit()* will not perform a cleanup. It will not close any files opened by the thread and files will remain in the open status even after the thread terminates. Calling *pthread_join* at the end of the main thread is the best way to achieve synchronisation and proper cleanup. The main thread, after finishing its task waits for the completion of other threads, which were joined to it using the *pthread_join* call. With a *pthread_join* call, the main thread waits other threads, which were joined to it, and finally merges to the single main thread. If a new thread spawned by the main thread is still not joined to the main thread, it will be counted against the system's maximum thread limit. Improper cleanup will lead to the failure of new thread creation.

Win32 Threads Win32 threads are the threads supported by various flavours of Windows Operating Systems. The Win32 Application Programming Interface (Win32 API) libraries provide the standard set of Win32 thread creation and management functions. Win32 threads are created with the API

```
HANDLE    CreateThread(LPSECURITY_ATTRIBUTES    lpThreadAttributes,DWORD
dwStackSize,  LPTHREAD_START_ROUTINE  lpStartAddress,  LPVOID  lpParameter,
DWORD dwCreationFlags, LPDWORD lpThreadId );
```

The parameter *lpThreadAttributes* defines the security attributes for the thread and *dwStackSize* defines the stack size for the thread. These two parameters are not supported by the Windows CE/Embedded Compact Real-Time Operating Systems and it should be kept as NULL and 0 respectively in a *CreateThread* API Call. The other parameters are

lpStartAddress: Pointer to the function which is to be executed by the thread.

lpParameter: Parameter specifying an application-defined value that is passed to the thread routine.

dwCreationFlags: Defines the state of the thread when it is created. Usually it is kept as 0 or CREATE_SUSPENDED implying the thread is created and kept at the suspended state.

lpThreadId: Pointer to a DWORD that receives the identifier for the thread.

On successful creation of the thread, *CreateThread* returns the handle to the thread and the thread identifier.

The API *GetCurrentThread(void)* returns the handle of the current thread and *GetCurrentThreadId(void)* returns its ID. *GetThreadPriority (HANDLE hThread)* API returns an integer value representing the current priority of the thread whose handle is passed as *hThread*. Threads are always created with normal priority (THREAD_PRIORITY_NORMAL. Refer MSDN documentation for the different thread priorities and their meaning). *SetThreadPriority (HANDLE hThread, int nPriority)* API is used for setting the priority of a thread. The first parameter to this function represents the thread handle and the second one the thread priority.

For Win32 threads, the normal thread termination happens when an exception occurs in the thread, or when the thread's execution is completed or when the primary thread or the process to which the thread is associated is terminated. A thread can exit itself by calling the *ExitThread (DWORD dwExitCode)* API. The parameter *dwExitCode* sets the exit code for thread termination. Calling *ExitThread* API frees all the resources utilised by the thread. The exit code of a thread can be checked by other threads by calling the *GetExitCodeThread (HANDLE hThread, LPDWORD lpExitCode)*. *TerminateThread (HANDLE hThread, DWORD dwExitCode)* API is used for terminating a thread from another thread. The handle *hThread* indicates which thread is to be terminated and *dwExitCode* sets the exit code for the thread. This API will not execute the thread termination and clean up code and may not free the resources occupied by the thread. *TerminateThread* is a potentially dangerous call and it should not be used in normal conditions as a mechanism for terminating a thread. Use this call only as a final choice. When a thread is terminated through *TerminateThread* method, the system releases the thread's initial stack and the thread will not get a chance to execute any user-mode code. Also any dynamic link libraries (dlls) attached to the thread are not notified that the thread is terminating. *TerminateThread* can lead to potential issues like: Non-releasing of the critical section object, any, owned by the thread, non-releasing of heap lock, if the thread is allocating memory from the heap, inconsistencies of the kernel32 state for the thread's process if the thread was executing certain kernel32 call when it is terminated, issues in shared dll functions, if the thread was manipulating the global state of a shared dll when it is terminated etc. *SuspendThread(HANDLE hThread)* API can be used for suspending a thread from execution provided the *handle hThread* possesses *THREAD_SUSPEND_RESUME* access right. If the *SuspendThread* API call succeeds, the thread stops executing and increments its internal suspend count. The thread becomes suspended if its suspend count is greater than zero. The *SuspendThread* function is primarily designed for use by debuggers. One must be cautious in using this API for the reason it may cause *deadlock* condition if the thread is suspended at a stage where it acquired a mutex or shared resource and another thread tries to access the same. The *ResumeThread(HANDLE hThread)* API is used for resuming a suspended thread. The *ResumeThread* API checks the suspend count of the specified thread. A suspend count of zero indicates that the specified thread is not currently in the suspended mode. If the count is not zero, the count is decremented by one and if the resulting count value is zero, the thread is resumed. The API *Sleep (DWORD* *dwMilliseconds)* can be used for suspending a thread for the duration specified in milliseconds by the *Sleep* function. The *Sleep* call is initiated by the thread.

Example 2

Write a multithreaded application using Win32 APIs to set up a counter in the main thread and secondary thread to count from 0 to 10 and print the counts from both the threads. Put a delay of 500 ms in between the successive printing in both the threads.

```
#include "stdafx.h"
#include "windows.h"
#include "stdio.h"
//*******************************************************************
//Child thread
//*******************************************************************
void ChildThread(void)
{
 char i;
 for (i = 0; i <= 10; ++i)
 {
  printf("Executing Child Thread : Counter = %d\n", i);
  Sleep(500);
 }
}
//*******************************************************************
//Primary thread
//*******************************************************************
int main(int argc, char* argv[])
{
 HANDLE hThread;
 DWORD dwThreadID;
 char i;
 hThread = CreateThread(NULL, 1000, (LPTHREAD_START_ROUTINE)ChildThread,
                        NULL, 0, &dwThreadID);
 if (hThread == NULL)
 {
  printf("Thread Creation Failed\nError No : %d\n", GetLastError());
  return 1;
 }
 for (i = 0; i <= 10; ++i)
 {
  printf("Executing Main Thread : Counter = %d\n", i);
  Sleep(500);
 }
 return 0;
}
```

To execute this program, create a new Win32 Console Application with Microsoft Visual Studio using Visual C++ and add the above piece of code to it and compile. The output obtained on running this application on a machine with Windows 10 operating system is given in Fig. 10.9.

```
Executing Main Thread: Counter = 0
Executing Child Thread:Counter = 0
Executing Main Thread: Counter = 1
Executing Child Thread:Counter = 1
Executing Main Thread: Counter = 2
Executing Child Thread:Counter = 2
Executing Main Thread: Counter = 3
Executing Child Thread:Counter = 3
Executing Main Thread: Counter = 4
Executing Child Thread:Counter = 4
Executing Main Thread: Counter = 5
Executing Child Thread:Counter = 5
Executing Child Thread:Counter = 6
Executing Main Thread: Counter = 6
Executing Child Thread:Counter = 7
Executing Main Thread: Counter = 7
Executing Child Thread:Counter = 8
Executing Main Thread: Counter = 8
Executing Child Thread:Counter = 9
Executing Main Thread: Counter = 9
Executing Child Thread:Counter = 10
Executing Main Thread: Counter = 10
```

Fig. 10.9 Output of the Win32 Multithreaded application

If you examine the output, you can see the switching between main and child threads. The output need not be the same always. The output is purely dependent on the scheduling policies implemented by the windows operating system for thread scheduling. You may get the same output or a different output each time you run the application.

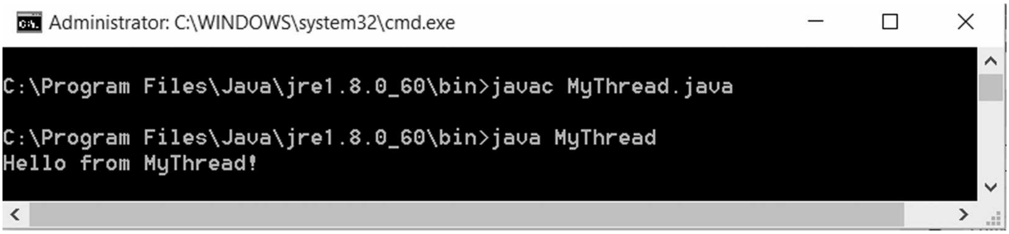
Java Threads Java threads are the threads supported by Java programming Language. The java thread class '*Thread*' is defined in the package '*java.lang*'. This package needs to be imported for using the thread creation functions supported by the Java thread class. There are two ways of creating threads in Java: Either by extending the base '*Thread*' class or by implementing an interface. Extending the thread class allows inheriting the methods and variables of the parent class (Thread class) only whereas interface allows a way to achieve the requirements for a set of classes. The following piece of code illustrates the implementation of Java threads with extending the thread base class '*Thread*'.

```
import java.lang.*;
public class MyThread extends Thread
{
	public void run()
	{
		System.out.println(“Hello from MyThread!”);
	}
	public static void main(String args[])
	{
		(new MyThread()).start();
	}
}
```

The above piece of code creates a new class *MyThread* by extending the base class *Thread*. It also overrides the *run()* method inherited from the base class with its own *run()* method. The *run()* method of *MyThread* implements all the task for the *MyThread* thread. The method *start()* moves the thread to a pool of threads waiting for their turn to be picked up for execution by the scheduler. The thread is said to be in the 'Ready' state at this stage. The scheduler picks the threads for execution from the pool based on the thread priorities.

```
E.g.  MyThread.start();
```

The output of the above piece of code when executed on Windows 10 platform is given in Fig. 10.10.



```
C:\Program Files\Java\jre1.8.0_60\bin>javac MyThread.java

C:\Program Files\Java\jre1.8.0_60\bin>java MyThread
Hello from MyThread!
```

Fig. 10.10 Output of the Java Multithreaded application

Invoking the static method *yield()* voluntarily give up the execution of the thread and the thread is moved to the pool of threads waiting to get their turn for execution, i.e. the thread enters the 'Ready' state.

```
E.g.  MyThread.yield();
```

The static method *sleep()* forces the thread to sleep for the duration mentioned by the sleep call, i.e. the thread enters the 'Suspend' mode. Once the sleep period is expired, the thread is moved to the pool of threads waiting to get their turn for execution, i.e. the thread enters the 'Ready' state. The method *sleep()* only guarantees that the thread will sleep for the minimum period mentioned by the argument to the call. It will not guarantee anything on the resume of the thread after the sleep period. It is dependent on the scheduler.

```
E.g.  MyThread.sleep(100); Sleep for 100 milliseconds.
```

Calling a thread Object's *wait()* method causes the thread object to wait. The thread will remain in the 'Wait' state until another thread invokes the *notify()* or *notifyAll()* method of the thread object which is waiting. The thread enters the 'Blocked' state when waiting for input from I/O devices or waiting for object lock in case of accessing shared resources. The thread is moved to the 'Ready' state on receiving the I/O input or on acquiring the object lock. The thread enters the 'Finished/Dead' state on completion of the task assigned to it or when the *stop()* method is explicitly invoked. The thread may also enter this state if it is terminated by an unrecoverable error condition.

For more information on Java threads, visit Sun Micro System's tutorial on Threads, available at http://java.sun.com/tutorial/applet/overview/threads.html

Summary So far we discussed about the various thread classes available for creation and management of threads in a multithreaded system in a General Purpose Operating System's perspective. From an RTOS perspective, POSIX threads and Win32 threads are the most commonly used thread class libraries for thread creation and management. Many non-standard, proprietary thread classes are also used by some proprietary RTOS. Portable threads (*Pth*), a very portable POSIX/ANSI-C based library from GNU, may be the "next generation" threads library. *Pth* provides non-preemptive priority based scheduling for multiple threads inside event driven applications. Visit http://www.gnu.org/software/pth/ for more details on GNU Portable threads.

10.3.2.3 Thread Pre-emption

Thread pre-emption is the act of pre-empting the currently running thread (stopping the currently running thread temporarily). Thread pre-emption ability is solely dependent on the Operating System. Thread pre-emption is performed for sharing the CPU time among all the threads. The execution switching among threads are known as '*Thread context switching*'. Thread context switching is dependent on the Operating system's scheduler and the type of the thread. When we say 'Thread', it falls into any one of the following types.

User Level Thread 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.

Kernel/System Level Thread 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.

For user level threads, 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. On the other hand, 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. There are many ways for binding user level threads with system/kernel level threads. The following section gives an overview of various thread binding models.

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 given under the POSIX thread library section is an illustrative example for application with Many-to-One thread model.

One-to-One Model In One-to-One model, each user level thread is bonded to a kernel/system level thread. Windows NT and Linux threads are examples for One-to-One thread models. The modified '*PThread*' example given under the '*Thread Pre-emption*' section 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.

10.3.2.4 Thread v/s Process

I hope, by now you got a reasonably good knowledge of *process* and *threads*. Now let us summarise the properties of *process* and *threads*.

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 heap 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 (shares 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 dies.

10.4 MULTIPROCESSING AND MULTITASKING

LO 4 Understand the difference between multiprocessing and multitasking

The terms *multiprocessing* and *multitasking* are a little confusing and sounds alike. In the operating system context *multiprocessing* describes the ability to execute multiple processes simultaneously. Systems which are capable of performing multiprocessing, are known as *multiprocessor* systems. *Multiprocessor* systems possess multiple CPUs and can execute multiple processes simultaneously.

The ability of the operating system to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*. In a uniprocessor system, it is not possible to execute multiple processes simultaneously. However, it is possible for a uniprocessor system to achieve some degree of pseudo parallelism in the execution of multiple processes by switching the execution among different processes. The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as *multitasking*. Multitasking creates the illusion of multiple tasks executing in parallel. Multitasking involves the switching of CPU from executing one task to another. In an earlier section '*The Structure of a Process*' of this chapter, we learned that a Process is identical to the physical processor in the sense it has own register set which mirrors the CPU registers, stack and Program Counter (PC). Hence, a '*process*' is considered as a '*Virtual processor*', awaiting its turn to have its properties switched into the physical processor. In a multitasking environment, when task/process switching happens, the virtual processor (task/process) gets its properties converted into that of the physical processor. The switching of the virtual processor to physical processor is controlled by the scheduler of the OS kernel. Whenever a CPU switching happens, the current context of execution should be saved to retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching. The context saving and retrieval is essential for resuming a process exactly from the point where it was interrupted due to CPU switching. The act of switching CPU among the processes or changing the current execution context is known as '*Context switching*'. The act of saving the current context which contains the context details (Register details, memory details, system resource usage details, execution details, etc.) for the currently running process at the time of CPU switching is known as '*Context saving*'. The process of retrieving the saved context details for a process, which is going to be executed due to CPU switching, is known as '*Context retrieval*'. Multitasking involves '*Context switching*' (Fig. 10.11), '*Context saving*' and '*Context retrieval*'.

Toss Juggling The skilful object manipulation game is a classic real world example for the multitasking illusion. The juggler uses a number of objects (balls, rings, etc.) and throws them up and catches them. At any point of time, he throws only one ball and catches only one per hand. However, the speed at which he is switching the balls for throwing and catching creates the illusion, he is throwing and catching multiple balls or using more than two hands ☺ simultaneously, to the spectators.

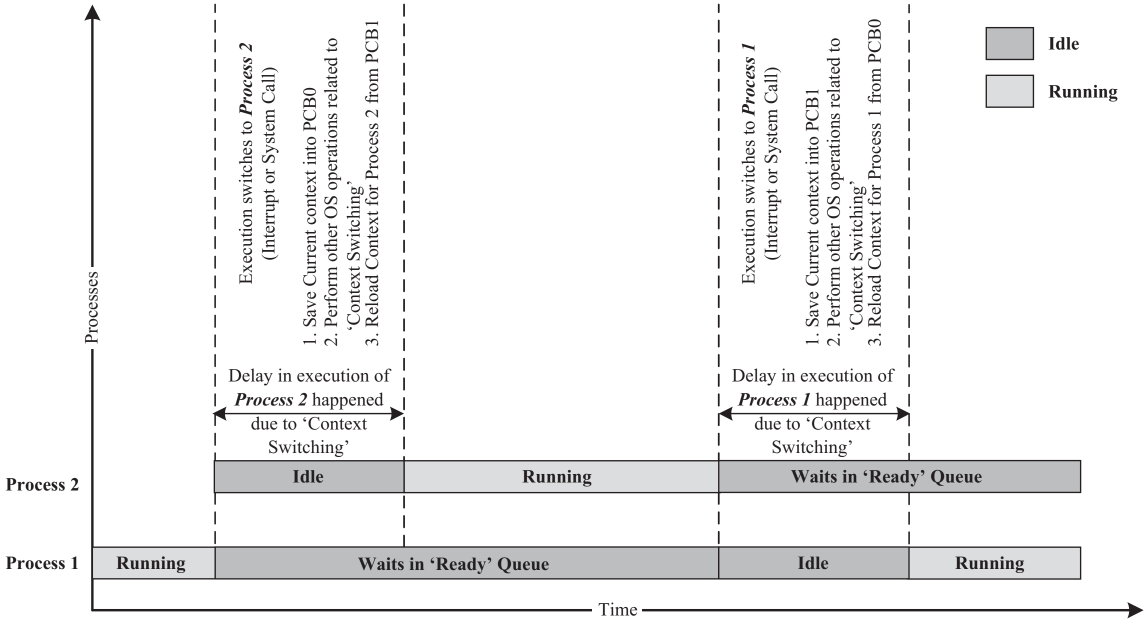


Fig. 10.11 Context switching

10.4.1 Types of Multitasking

As we discussed earlier, multitasking involves the switching of execution among multiple tasks. Depending on how the switching act is implemented, multitasking can be classified into different types. The following section describes the various types of multitasking existing in the Operating System's context.

10.4.1.1 Co-operative Multitasking

Co-operative multitasking is the most primitive form of multitasking in which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU. In this method, any task/process can hold the CPU as much time as it wants. Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, it is known as co-operative multitasking. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU.

10.4.1.2 Preemptive Multitasking

Preemptive multitasking ensures that every task/process gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling. As the name indicates, in preemptive multitasking, the currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/process priority.

10.4.1.3 Non-preemptive Multitasking

In non-preemptive multitasking, the process/task, which is currently given the CPU time, is allowed to execute until it terminates (enters the '*Completed*' state) or enters the '*Blocked/Wait*' state, waiting for an I/O or system resource. The co-operative and non-preemptive multitasking differs in their behaviour when they are in the '*Blocked/Wait*' state. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the '*Blocked/Wait*' state, waiting for an I/O, or a shared resource access or an event to occur whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O or system resource or an event to occur.

10.5 TASK SCHEDULING

LO 5 Describe the FCFS/FIFO, LCFS/LIFO, SJF and priority based task/process scheduling

As we already discussed, 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. Task scheduling forms the basis of multitasking. 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*'. 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. 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.

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: 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 a minimal for a good scheduling algorithm.

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.