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.

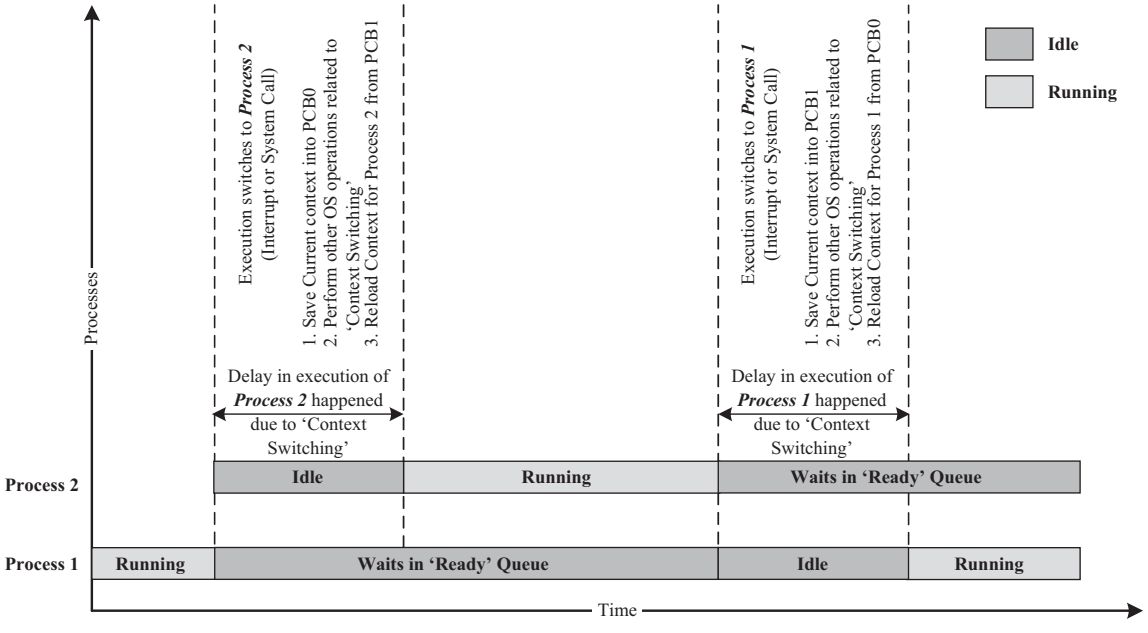


Fig. 10.11 Context switching

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.

The Operating System maintains various queues† in connection with the CPU scheduling, and a process passes through these queues during the course of its admittance to execution completion.

The various queues maintained by OS in association with CPU scheduling are:

Job Queue: 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**.**

A process migrates through all these queues during its journey from '*Admitted*' to '*Completed*' stage. The following diagrammatic representation (Fig. 10.12) illustrates the transition of a process through the various queues.

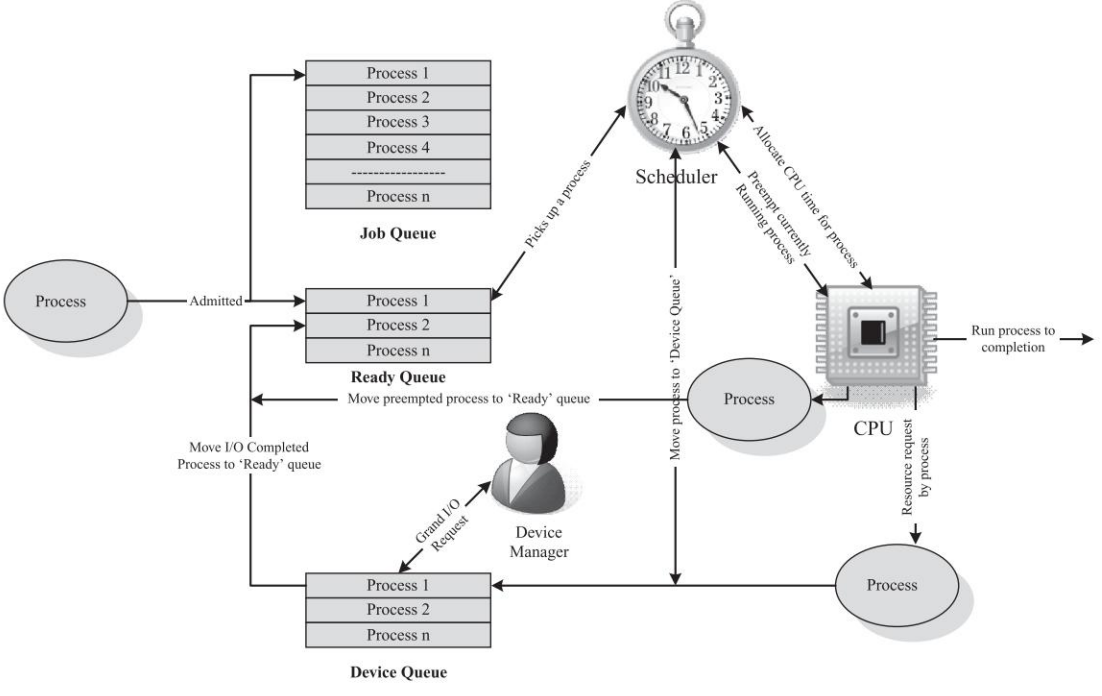


Fig. 10.12 Illustration of process transition through various queues

Based on the scheduling algorithm used, the scheduling can be classified into the following categories.

10.5.1 Non-preemptive Scheduling

Non-preemptive scheduling is employed in systems, which implement non-preemptive multitasking model. In this scheduling type, 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. The various types of non-preemptive scheduling adopted in task/process scheduling are listed below.

† Queue is a special kind of arrangement of a collection of objects. In the operating system context queue is considered as a buffer.

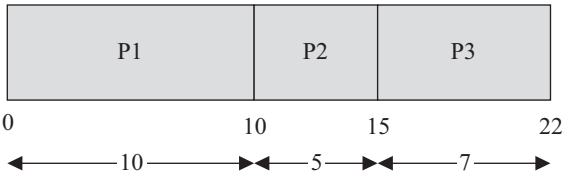
10.5.1.1 First-Come-First-Served (FCFS)/ FIFO Scheduling

As the name indicates, the First-Come-First-Served (FCFS) scheduling algorithm allocates CPU time to the processes based on the order in which they enter the '*Ready*' queue. The first entered process is serviced first. It is same as any real world application where queue systems are used; e.g. Ticketing reservation system where people need to stand in a queue and the first person standing in the queue is serviced first. FCFS scheduling is also known as First In First Out (FIFO) where the process which is put first into the '*Ready*' queue is serviced first.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters 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).

The sequence of execution of the processes by the CPU is represented as



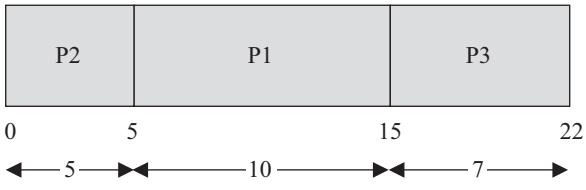
Assuming the CPU is readily available at the time of arrival of P1, P1 starts executing without any waiting in the '*Ready*' queue. Hence the waiting time for P1 is zero. The waiting time for all processes are given as

Waiting Time for P1 = 0 ms (P1 starts executing first)
Waiting Time for P2 = 10 ms (P2 starts executing after completing P1)
Waiting Time for P3 = 15 ms (P3 starts executing after completing P1 and P2)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P2+P3)) / 3
= (0+10+15)/3 = 25/3
= 8.33 milliseconds
Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 15 ms (-Do-)
Turn Around Time (TAT) for P3 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P1+P2+P3)) / 3
= (10+15+22)/3 = 47/3
= 15.66 milliseconds
Average Turn Around Time (TAT) is the sum of average waiting time and average execution time.
Average Execution Time = (Execution time for all processes)/No. of processes
= (Execution time for (P1+P2+P3))/3
= (10+5+7)/3 = 22/3
= 7.33
Average Turn Around Time = Average waiting time + Average execution time
= 8.33 + 7.33
= 15.66 milliseconds

Example 2

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) for the above example if the process enters the 'Ready' queue together in the order P2, P1, P3.

The sequence of execution of the processes by the CPU is represented as



Assuming the CPU is readily available at the time of arrival of P2, P2 starts executing without any waiting in the '*Ready*' queue. Hence the waiting time for P2 is zero. The waiting time for all processes is given as

Waiting Time for P2 = 0 ms (P2 starts executing first)
Waiting Time for P1 = 5 ms (P1 starts executing after completing P2)
Waiting Time for P3 = 15 ms (P3 starts executing after completing P2 and P1)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P2+P1+P3)) / 3
= (0+5+15)/3 = 20/3
= 6.66 milliseconds
Turn Around Time (TAT) for P2 = 5 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P1 = 15 ms (-Do-)
Turn Around Time (TAT) for P3 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P2+P1+P3)) / 3
= (5+15+22)/3 = 42/3
= 14 milliseconds

The Average waiting time and Turn Around Time (TAT) depends on the order in which the processes enter the '*Ready*' queue, regardless there estimated completion time.

From the above two examples it is clear that the Average waiting time and Turn Around Time improve if the process with shortest execution completion time is scheduled first.

The major drawback of FCFS algorithm is that it favours monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task. If the process contains any I/O operation, the CPU is relinquished by the process. In general, FCFS favours CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilisation. The average waiting time is not minimal for FCFS scheduling algorithm.

10.5.1.2 Last-Come-First Served (LCFS)/LIFO Scheduling

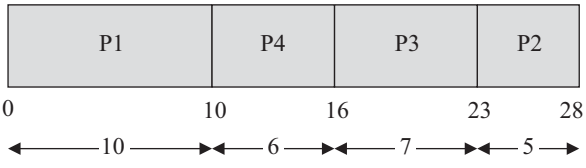
The Last-Come-First Served (LCFS) scheduling algorithm also allocates CPU time to the processes based on the order in which they are entered in the '*Ready*' queue. The last entered process is serviced first. LCFS scheduling is also known as Last In First Out (LIFO) where the process, which is put last into the '*Ready*' queue, is serviced first.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3 (Assume only P1 is present in the '*Ready*' queue when

the scheduler picks it up and P2, P3 entered '*Ready*' queue after that). Now a new process P4 with estimated completion time 6 ms enters the 'Ready' queue after 5 ms of scheduling 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). Assume all the processes contain only CPU operation and no I/O operations are involved.

Initially there is only P1 available in the Ready queue and the scheduling sequence will be P1, P3, P2. P4 enters the queue during the execution of P1 and becomes the last process entered the '*Ready*' queue. Now the order of execution changes to P1, P4, P3, and P2 as given below.



The waiting time for all the processes is given as
Waiting Time for P1 = 0 ms (P1 starts executing first)
Waiting Time for P4 = 5 ms (P4 starts executing after completing P1. But P4 arrived after 5 ms of execution of P1. Hence its waiting time = Execution start time – Arrival Time = 10 – 5 = 5)
Waiting Time for P3 = 16 ms (P3 starts executing after completing P1 and P4)
Waiting Time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (0 + 5 + 16 + 23)/4 = 44/4
= 11 milliseconds
Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 11 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (10 – 5) + 6 = 5 + 6)
Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P1+P4+P3+P2)) / 4
= (10+11+23+28)/4 = 72/4
= 18 milliseconds

LCFS scheduling is not optimal and it also possesses the same drawback as that of FCFS algorithm.

10.5.1.3 Shortest Job First (SJF) Scheduling

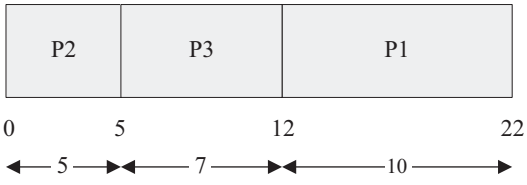
Shortest Job First (SJF) scheduling algorithm 'sorts the '*Ready*' queue' each time a process relinquishes the CPU (either the process terminates or enters the '*Wait*' state waiting for I/O or system resource) to pick the process with shortest (least) estimated completion/run time. In SJF, the process with the shortest estimated run time is scheduled first, followed by the next shortest process, and so on.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. 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 SJF algorithm.

The scheduler sorts the '*Ready*' queue based on the shortest estimated completion time and schedules the process with the least estimated completion time first and the next least one as second, and so on. The order in which the processes are scheduled for execution is represented as



The estimated execution time of P2 is the least (5 ms) followed by P3 (7 ms) and P1 (10 ms).
The waiting time for all processes are given as
Waiting Time for P2 = 0 ms (P2 starts executing first)
Waiting Time for P3 = 5 ms (P3 starts executing after completing P2)
Waiting Time for P1 = 12 ms (P1 starts executing after completing P2 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P2+P3+P1)) / 3
= (0+5+12)/3 = 17/3
= 5.66 milliseconds
Turn Around Time (TAT) for P2 = 5 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P3 = 12 ms (-Do-)
Turn Around Time (TAT) for P1 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P2+P3+P1)) / 3
= (5+12+22)/3 = 39/3
= 13 milliseconds
Average Turn Around Time (TAT) is the sum of average waiting time and average execution time.
The average Execution time = (Execution time for all processes)/No. of processes
= (Execution time for (P1+P2+P3))/3
= (10+5+7)/3 = 22/3 = 7.33
Average Turn Around Time = Average Waiting time + Average Execution time
= 5.66 + 7.33
= 13 milliseconds
From this example, it is clear that the average waiting time and turn around time is much improved with the SJF scheduling for the same processes when compared to the FCFS algorithm.

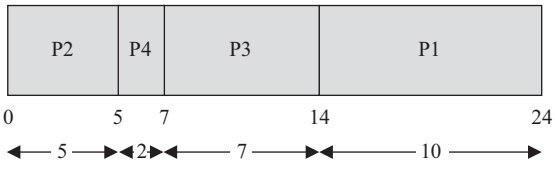
Example 2

Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time for the above example if a new process P4 with estimated completion time 2 ms enters the 'Ready' queue after 2 ms of execution of P2. Assume all the processes contain only CPU operation and no I/O operations are involved.

At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the SJF scheduler picks up the process with the least execution completion time (In this example P2 with

execution completion time 5 ms) for scheduling. The execution sequence diagram for this is same as that of Example 1.

Now process P4 with estimated execution completion time 2 ms enters the '*Ready*' queue after 2 ms of start of execution of P2. Since the SJF algorithm is non-preemptive and process P2 does not contain any I/O operations, P2 continues its execution. After 5 ms of scheduling, P2 terminates and now the scheduler again sorts the '*Ready*' queue for process with least execution completion time. Since the execution completion time for P4 (2 ms) is less than that of P3 (7 ms), which was supposed to be run after the completion of P2 as per the '*Ready*' queue available at the beginning of execution scheduling, P4 is picked up for executing. Due to the arrival of the process P4 with execution time 2 ms, the '*Ready*' queue is re-sorted in the order P2, P4, P3, P1. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram



The waiting time for all the processes are given as

Waiting time for P2 = 0 ms (P2 starts executing first)

Waiting time for P4 = 3 ms (P4 starts executing after completing P2. But P4 arrived after 2 ms of execution of P2. Hence its waiting time = Execution start time – Arrival Time = 5 – 2 = 3)

Waiting time for P3 = 7 ms (P3 starts executing after completing P2 and P4)

Waiting time for P1 = 14 ms (P1 starts executing after completing P2, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P2+P4+P3+P1)) / 4
= (0 + 3 + 7 + 14)/4 = 24/4
= 6 milliseconds

Turn Around Time (TAT) for P2 = 5 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 5 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (5 – 2) + 2 = 3 + 2)

Turn Around Time (TAT) for P3 = 14 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P1 = 24 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all Processes) / No. of Processes
= (Turn Around Time for (P2+P4+P3+P1)) / 4
= (5+5+14+24)/4 = 48/4
= 12 milliseconds

The average waiting time for a given set of process is minimal in SJF scheduling and so it is optimal compared to other non-preemptive scheduling like FCFS. The major drawback of SJF algorithm is that a process whose estimated execution completion time is high may not get a chance to execute if more and more processes with least estimated execution time enters the '*Ready*' queue before the process with longest estimated execution time started its execution (In non-preemptive SJF). This condition is known as '*Starvation*'. Another drawback of SJF is that it is difficult to know in advance the next shortest process in the '*Ready*' queue for scheduling since new processes with different estimated execution time keep entering the '*Ready*' queue at any point of time.

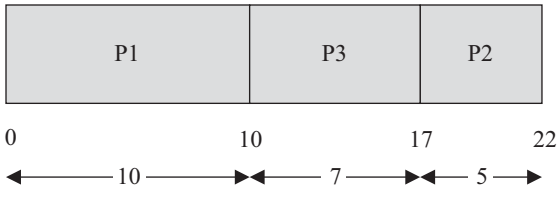
10.5.1.4 Priority Based Scheduling

The Turn Around Time (TAT) and waiting time for processes in non-preemptive scheduling varies with the type of scheduling algorithm. Priority based non-preemptive scheduling algorithm ensures that a process with high priority is serviced at the earliest compared to other low priority processes in the 'Ready' queue. The priority of a task/process can be indicated through various mechanisms. The Shortest Job First (SJF) algorithm can be viewed as a priority based scheduling where each task is prioritised in the order of the time required to complete the task. The lower the time required for completing a process the higher is its priority in SJF algorithm. Another way of priority assigning is associating a priority to the task/process at the time of creation of the task/process. The priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent. For Example, Windows CE supports 256 levels of priority (0 to 255 priority numbers). 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 variation from high to low is represented by numbers from 0 to the maximum priority or by numbers from maximum priority to 0. For Windows CE operating system a priority number 0 indicates the highest priority and 255 indicates the lowest priority. This convention need not be universal and it depends on the kernel level implementation of the priority structure. The non-preemptive priority based scheduler sorts the 'Ready' queue based on priority and picks the process with the highest level of priority for execution.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 0, 3, 2 (0—highest priority, 3—lowest priority) respectively enters the ready queue together. 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.

The scheduler sorts the '*Ready*' queue based on the priority and schedules the process with the highest priority (P1 with priority number 0) first and the next high priority process (P3 with priority number 2) as second, and so on. The order in which the processes are scheduled for execution is represented as



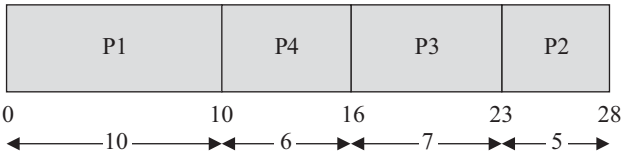
The waiting time for all the processes are given as
Waiting time for P1 = 0 ms (P1 starts executing first)
Waiting time for P3 = 10 ms (P3 starts executing after completing P1)
Waiting time for P2 = 17 ms (P2 starts executing after completing P1 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P3+P2)) / 3
= (0+10+17)/3 = 27/3
= 9 milliseconds
Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P3 = 17 ms (-Do-)
Turn Around Time (TAT) for P2 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes

= (Turn Around Time for (P1+P3+P2)) / 3
= (10+17+22)/3 = 49/3
= 16.33 milliseconds

Example 2

Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time for the above example if a new process P4 with estimated completion time 6 ms and priority 1 enters the 'Ready' queue after 5 ms of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the scheduler picks up the process with the highest priority (In this example P1 with priority 0) for scheduling. The execution sequence diagram for this is same as that of Example 1. Now process P4 with estimated execution completion time 6 ms and priority 1 enters the '*Ready*' queue after 5 ms of execution of P1. Since the scheduling algorithm is non-preemptive and process P1 does not contain any I/O operations, P1 continues its execution. After 10 ms of scheduling, P1 terminates and now the scheduler again sorts the '*Ready*' queue for process with highest priority. Since the priority for P4 (priority 1) is higher than that of P3 (priority 2), which was supposed to be run after the completion of P1 as per the '*Ready*' queue available at the beginning of execution scheduling, P4 is picked up for executing. Due to the arrival of the process P4 with priority 1, the '*Ready*' queue is resorted in the order P1, P4, P3, P2. At the beginning it was P1, P3, P2. The execution sequence now changes as per the following diagram



The waiting time for all the processes are given as
Waiting time for P1 = 0 ms (P1 starts executing first)
Waiting time for P4 = 5 ms (P4 starts executing after completing P1. But P4 arrived after 5 ms of execution of P1. Hence its waiting time = Execution start time – Arrival Time = 10 – 5 = 5)
Waiting time for P3 = 16 ms (P3 starts executing after completing P1 and P4)
Waiting time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (0 + 5 + 16 + 23)/4 = 44/4
= 11 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 11 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (10 – 5) + 6 = 5 + 6)
Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P2 + P4 + P3 + P1)) / 4
= (10 + 11 + 23 + 28)/4 = 72/4
= 18 milliseconds

Similar to SJF scheduling algorithm, non-preemptive priority based algorithm also possess the drawback of '*Starvation*' where a process whose priority is low may not get a chance to execute if more and more processes with higher priorities enter the '*Ready*' queue before the process with lower priority started its execution. '*Starvation*' can be effectively tackled in priority based non-preemptive scheduling by dynamically raising the priority of the low priority task/process which is under starvation (waiting in the ready queue for a longer time for getting the CPU time). The technique of gradually raising the priority of processes which are waiting in the 'Ready' queue as time progresses, for preventing '*Starvation*', is known as '*Aging*'.

10.5.2 Preemptive Scheduling

Preemptive scheduling is employed in systems, which implements preemptive multitasking model. In preemptive scheduling, every task in the '*Ready*' queue gets a chance to execute. When and how often each process gets a chance to execute (gets the CPU time) is dependent on the type of preemptive scheduling algorithm used for scheduling the processes. In this kind of scheduling, the scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the '*Ready*' queue for execution. 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 can be implemented in different approaches. The two important approaches adopted in preemptive scheduling are time-based preemption and priority-based preemption. The various types of preemptive scheduling adopted in task/process scheduling are explained below.

10.5.2.1 Preemptive SJF Scheduling/Shortest Remaining Time (SRT)

The non-preemptive SJF scheduling algorithm sorts the 'Ready' queue only after completing the execution of the current process or when the process enters 'Wait' state, whereas 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.

Now let us solve Example 2 given under the Non-preemptive SJF scheduling for preemptive SJF scheduling. The problem statement and solution is explained in the following example.

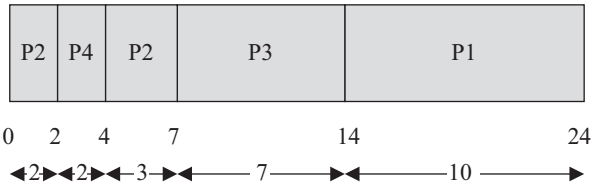
Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters 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.

At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the SRT scheduler picks up the process with the shortest remaining time for execution completion (In this example, P2 with remaining time 5 ms) for scheduling. The execution sequence diagram for this is same as that of example 1 under non-preemptive SJF scheduling.

Now process P4 with estimated execution completion time 2 ms enters the '*Ready*' queue after 2 ms of

start of execution of P2. Since the SRT algorithm is preemptive, the remaining time for completion of process P2 is checked with the remaining time for completion of process P4. The remaining time for completion of P2 is 3 ms which is greater than that of the remaining time for completion of the newly entered process P4 (2 ms). Hence P2 is preempted and P4 is scheduled for execution. P4 continues its execution to finish since there is no new process entered in the 'Ready' queue during its execution. After 2 ms of scheduling P4 terminates and now the scheduler again sorts the '*Ready*' queue based on the remaining time for completion of the processes present in the 'Ready' queue. Since the remaining time for P2 (3 ms), which is preempted by P4 is less than that of the remaining time for other processes in the 'Ready' queue, P2 is scheduled for execution. Due to the arrival of the process P4 with execution time 2 ms, the '*Ready*' queue is re-sorted in the order P2, P4, P2, P3, P1. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram



The waiting time for all the processes are given as

Waiting time for P2 = 0 ms + (4 – 2) ms = 2 ms (P2 starts executing first and is interrupted by P4 and has to wait till the completion of P4 to get the next CPU slot)

Waiting time for P4 = 0 ms (P4 starts executing by preempting P2 since the execution time for completion of P4 (2 ms) is less than that of the Remaining time for execution completion of P2 (Here it is 3 ms))

Waiting time for P3 = 7 ms (P3 starts executing after completing P4 and P2)

Waiting time for P1 = 14 ms (P1 starts executing after completing P4, P2 and P3)

Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P4+P2+P3+P1)) / 4
= (0 + 2 + 7 + 14)/4 = 23/4
= 5.75 milliseconds

Turn Around Time (TAT) for P2 = 7 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 2 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (2 – 2) + 2)

Turn Around Time (TAT) for P3 = 14 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P1 = 24 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P2+P4+P3+P1)) / 4
= (7+2+14+24)/4 = 47/4
= 11.75 milliseconds

Now let's compare the Average Waiting time and Average Turn Around Time with that of the Average waiting time and Average Turn Around Time for non-preemptive SJF scheduling (Refer to Example 2 given under the section Non-preemptive SJF scheduling)

Average Waiting Time in non-preemptive SJF scheduling = 6 ms

Average Waiting Time in preemptive SJF scheduling = 5.75 ms

Average Turn Around Time in non-preemptive SJF scheduling = 12 ms

Average Turn Around Time in preemptive SJF scheduling = 11.75 ms

This reveals that the Average waiting Time and Turn Around Time (TAT) improves significantly with preemptive SJF scheduling.

10.5.2.2 Round Robin (RR) Scheduling

The term *Round Robin* is very popular among the sports and games activities. You might have heard about 'Round Robin' league or 'Knock out' league associated with any football or cricket tournament. In the 'Round Robin' league each team in a group gets an equal chance to play against the rest of the teams in the same group whereas in the 'Knock out' league the losing team in a match moves out of the tournament ☺.

In the process scheduling context also, '*Round Robin*' brings the same message "Equal chance to all". In Round Robin scheduling, each process in the 'Ready' queue is executed for a pre-defined time slot. The execution starts with picking up the first process in the 'Ready' queue (see Fig. 10.13). 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. This reveals that the Round Robin scheduling is similar to the FCFS scheduling and the only difference is that a time slice based preemption is added to switch the execution between the processes in the 'Ready' queue. 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.

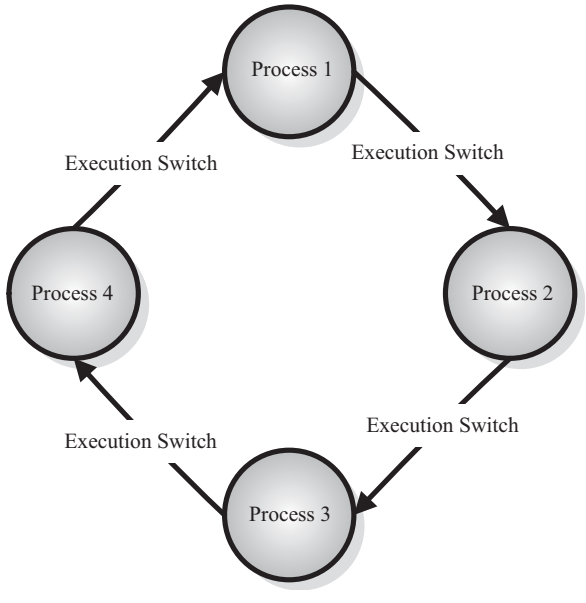


Fig. 10.13 Round Robin Scheduling

The time slice is provided by the *timer tick* feature of the time management unit of the OS kernel (Refer the Time management section under the subtopic '*The Real-Time kernel*' for more details on Timer tick). Time slice is kernel dependent and it varies in the order of a few microseconds to milliseconds. Certain OS kernels may allow the time slice as user configurable. 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

FCFS policy (That is, a process entering the Ready queue first gets its fixed execution time first and so on…). 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. The implementation of RR scheduling is kernel dependent. The following code snippet illustrates the RR scheduling implementation for RTX51 Tiny OS, an 8bit OS for *8051* microcontroller from Keil Software (www.keil.com), an ARM® Company.

```
#include <rtx51tny.h>   /* Definitions for RTX51 Tiny */
int counter0;
int counter1;

job0 () _task_ 0 {
  os_create_task (1);   /* Mark task 1 as "ready"      */

  while (1)  {          /* Endless loop                */
    counter0++;         /* Increment counter 0         */
  }
}
job1 () _task_ 1 {
  while (1)  {          /* Endless loop                */
    counter1++;         /* Increment counter 1         */
  }
}
```

RTX51 defines the tasks as simple C functions with void return type and void argument list. The attribute *_task_* is used for declaring a function as task. The general form of declaring a task is

```
     void func (void) _task_ task_id
```

where *func* is the name of the task and *task_id* is the ID of the task. RTX51 supports up to 16 tasks and so *task_id* varies from 0 to 15. All tasks should be implemented as endless loops.

The two tasks in this program are counter loops. RTX51 Tiny starts executing task 0 which is the function named job0. This function creates another task called job1. After job0 executes for its time slice, RTX51 Tiny switches to job1. After job1 executes for its time slice, RTX51 Tiny switches back to job0. This process is repeated forever.

Now let's check how the RTX51 Tiny RR Scheduling can be implemented in an embedded device (A smart card reader) which addresses the following requirements.

- Check the presence of a card
- Process the data received from the card
- Update the Display
- Check the serial port for command/data
- Process the data received from serial port

These four requirements can be considered as four tasks. Implement them as four RTX51 tasks as explained below.

```
void check_card_task (void) _task_ 1
{
/* This task checks for the presence of a card */
/* Implement the necessary functionality here */
}
```

```
void process_card_task (void) _task_ 2
{
/* This task processes the data received from the card */
/* Implement the necessary functionality here */
}
void check_serial_io_task (void) _task_ 3
{
/* This task checks for serial I/O */
/* Implement the necessary functionality here */
}
void process_serial_data_task (void) _task_ 4
{
/* This task processes the data received from the serial port */
/* Implement the necessary functionality here */
}
```

Now the tasks are created. Next step is scheduling the tasks. The following code snippet illustrates the scheduling of tasks.

```
void startup_task (void) _task_ 0
{
os_create_task (1);    /* Create check_card_task Task */
os_create_task (2);    /* Create process_card_task Task */
os_create_task (3);    /* Create serial_io_task Task */

os_create_task (4);    /* Create serial_data_task Task */
os_delete_task (0);    /* Delete the Startup Task */
}
```

The *os_create_task* (task_ID) RTX51 Tiny kernel call puts the task with task ID *task_ID* in the 'Ready' state. All the ready tasks begin their execution at the next available opportunity. RTX51 Tiny does not have a *main ()* function to begin the code execution; instead it starts with executing task 0. Task 0 is used for creating other tasks. Once all the tasks are created, task 0 is stopped and removed from the task list with the *os_delete_task* kernel call. The RR scheduler selects each task based on the time slice and continues the execution. If we observe the tasks we can see that there is no point in executing the task *process_card_task* (Task 2) without detecting a card and executing the task *process_serial_data_task* (Task 4) without receiving some data in the serial port. In summary task 2 needs to be executed only when task 1 reports the presence of a card and task 4 needs to be executed only when task 3 reports the arrival of data at serial port. So these tasks (tasks 2 and 4) need to be put in the 'Ready' state only on satisfying these conditions. Till then these tasks can be put in the 'Wait' state so that the RR scheduler will not pick them for scheduling and the RR scheduling is effectively utilised among the other tasks. This can be achieved by implementing the wait and notify mechanism in the related tasks. Task 2 can be coded in a way that it waits for the card present event and task 1 signals the event 'card detected'. In a similar fashion Task 4 can be coded in such a way that it waits for the serial data received event and task 3 signals the reception of serial data on receiving serial data from serial port. The following code snippet explains the same.

```
void check_card_task (void) _task_ 1
{
/* This task checks for the presence of a card */
```

```
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the presence of card and card reading
//……
if (card is present)
//Signal card detected to task 2
os_send_signal (2)
}
}

void process_card_task (void) _task_ 2
{
/* This task processes the data received from the card */
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the signaling of card present event
os_wait1(K_SIG);
//Process card data
}
}

void check_serial_io_task (void) _task_ 3
{
/* This task checks for serial I/O */
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the reception of serial data
//……
if (data is received)
//Signal serial data reception to task 4
os_send_signal (4)
}
}

void process_serial_data_task (void) _task_ 4
{
/* This task processes the data received from the serial port */
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the signaling of serial data received event
os_wait1(K_SIG);
//Process card data
}
}
```

The *os_send_signal* (*Task ID*) kernel call sends a signal to task *Task ID*. If the specified task is already waiting for a signal, this function call readies the task for execution but does not start it. The *os_wait1* (*event*) kernel call halts the current task and waits for an event to occur. The *event* argument specifies the event to wait for and may have only the value K_SIG which waits for a signal. RTX51 uses the Timer 0 of *8051* for time slice generation. The time slice can be configured by the user by changing the time slice related parameters in the RTX51 Tiny OS configuration file **CONF_TNY.A51** file which is located in the **\Keil_v5\C51\RtxTiny2\SourceCode** folder. Configuration options in **CONF_TNY.A51** allow users to:

- Specify the Timer Tick Interrupt Register Bank.
- Specify the Timer Tick Interval (in *8051* machine cycles).
- Specify user code to execute in the Timer Tick Interrupt.
- Specify the Round-Robin Timeout.
- Enable or disable Round-Robin Task Switching.
- Specify that your application includes long duration interrupts.
- Specify whether or not code banking is used.
- Define the top of the RTX51 Tiny stack.
- Specify the minimum stack space required.
- Specify code to execute in the event of a stack error.
- Define idle task operations.

The RTX51 kernel provides a set of task management functions for managing the tasks. At any point of time each RTX51 task is exactly in any one of the following state.

Task State	State Description
RUNNING	The task that is currently running is in the RUNNING State. Only one task at a time may be in this state. The *os_running_task_id* kernel call returns the task number (ID) of the currently executing task.
READY	Tasks which are ready to run are in the READY State. Once the Running task has completed processing, RTX51 Tiny selects and starts the next Ready task. A task may be made ready immediately (even if the task is waiting for a timeout or signal) by setting its ready flag using the *os_set_ready or isr_set_ready* kernel functions.
WAITING	Tasks which are waiting for an event are in the WAITING State. Once the event occurs, the task is switched to the READY State. The *os_wait function* is used for placing a task in the WAITING State.
DELETED	Tasks which have not been started or tasks which have been deleted are in the DELETED State. The *os_delete_task* routine places a task that has been started (with *os_create_task*) into the DELETED State.
TIME-OUT	Tasks which were interrupted by a Round-Robin Time-Out are in the TIME-OUT State. This state is equivalent to the READY State for Round-Robin programs.

Refer the documentation available with RTX51 Tiny OS for more information on the various RTX51 task management kernel functions and their usage.

RR scheduling with interrupts is a good choice for the design of comparatively less complex *Real-Time Embedded Systems*. In this approach, the tasks which require less *Real-Time* attention can be scheduled with Round Robin scheduling and the tasks which require *Real-Time* attention can be scheduled through Interrupt Service Routines. RTX51 Tiny supports Interrupts with RR scheduling. For RTX51 the time slice for RR scheduling is provided by the Timer interrupt and if the interrupt is of high priority than that of the timer interrupt and if its service time (ISR) is longer than the timer tick interval, the RTX51 timer interrupt may

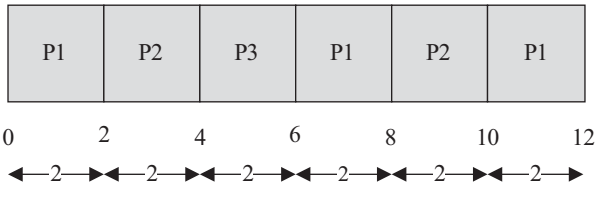
be interrupted by the ISR and it may be reentered by a subsequent RX51 Tiny timer interrupt. Hence proper care must be taken to limit the ISR time within the timer tick interval or to protect the timer tick interrupt code from reentrancy. Otherwise unexpected results may occur. The limitations of RR with interrupt generic approach are the limited number of interrupts supported by embedded processors and the interrupt latency happening due to the context switching overhead.

RR can also be used as technique for resolving the priority in scheduling among the tasks with same level of priority. We will discuss about how RR scheduling can be used for resolving the priority among equal tasks under the VxWorks kernel in a later chapter.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively, enters 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.

The scheduler sorts the '*Ready*' queue based on the FCFS policy and picks up the first process P1 from the 'Ready' queue and executes it for the time slice 2 ms. When the time slice is expired, P1 is preempted and P2 is scheduled for execution. The Time slice expires after 2ms of execution of P2. Now P2 is preempted and P3 is picked up for execution. P3 completes its execution within the time slice and the scheduler picks P1 again for execution for the next time slice. This procedure is repeated till all the processes are serviced. The order in which the processes are scheduled for execution is represented as



The waiting time for all the processes are given as

Waiting time for P1 = 0 + (6 – 2) + (10 – 8) = 0 + 4 + 2 = 6 ms
(P1 starts executing first and waits for two time slices to get execution back and again 1 time slice for getting CPU time)

Waiting time for P2 = (2 – 0) + (8 – 4) = 2 + 4 = 6 ms
(P2 starts executing after P1 executes for 1 time slice and waits for two time slices to get the CPU time)

Waiting time for P3 = (4 – 0) = 4 ms
(P3 starts executing after completing the first time slices for P1 and P2 and completes its execution in a single time slice)

Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P1 + P2 + P3)) / 3
= (6 + 6 + 4)/3 = 16/3
= 5.33 milliseconds

Turn Around Time (TAT) for P1 = 12 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 10 ms (-Do-)
Turn Around Time (TAT) for P3 = 6 ms (-Do-)

Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P1 + P2 + P3))/3
= (12 + 10 + 6)/3 = 28/3
= 9.33 milliseconds

Average Turn Around Time (TAT) is the sum of average waiting time and average execution time.

Average Execution time = (Execution time for all the process)/No. of processes
= (Execution time for (P1 + P2 + P3))/3
= (6 + 4 + 2)/3 = 12/3 = 4

Average Turn Around Time = Average Waiting time + Average Execution time
= 5.33 + 4
= 9.33 milliseconds

RR scheduling involves lot of overhead in maintaining the time slice information for every process which is currently being executed.

10.5.2.3 Priority Based Scheduling

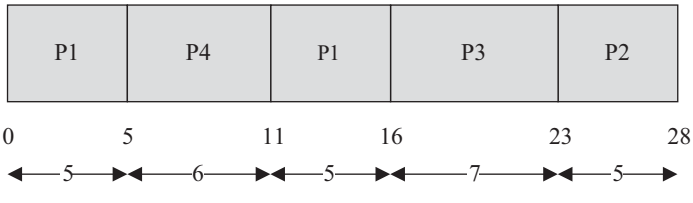
Priority based preemptive scheduling algorithm is same as that of the non-preemptive priority based scheduling except for the switching of execution between tasks. In preemptive scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the non-preemptive scheduling any high priority process entering the 'Ready' queue is scheduled only after the currently executing process completes its execution or only when it voluntarily relinquishes the CPU. The priority of a task/process in preemptive scheduling is indicated in the same way as that of the mechanism adopted for non-preemptive multitasking. Refer the non-preemptive priority based scheduling discussed in an earlier section of this chapter for more details.

Example 1

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 enters 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. Assume all the processes contain only CPU operation and no I/O operations are involved.

At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the scheduler picks up the process with the highest priority (In this example P1 with priority 1) for scheduling.

Now process P4 with estimated execution completion time 6 ms and priority 0 enters the '*Ready*' queue after 5 ms of start of execution of P1. Since the scheduling algorithm is preemptive, P1 is preempted by P4 and P4 runs to completion. After 6 ms of scheduling, P4 terminates and now the scheduler again sorts the '*Ready*' queue for process with highest priority. Since the priority for P1 (priority 1), which is preempted by P4 is higher than that of P3 (priority 2) and P2 ((priority 3), P1 is again picked up for execution by the scheduler. Due to the arrival of the process P4 with priority 0, the '*Ready*' queue is resorted in the order P1, P4, P1, P3, P2. At the beginning it was P1, P3, P2. The execution sequence now changes as per the following diagram



The waiting time for all the processes are given as
Waiting time for P1 = 0 + (11 – 5) = 0 + 6 = 6 ms
(P1 starts executing first and gets preempted by P4 after 5 ms and again gets the CPU time after completion of P4)
Waiting time for P4 = 0 ms
(P4 starts executing immediately on entering the 'Ready' queue, by preempting P1)
Waiting time for P3 = 16 ms (P3 starts executing after completing P1 and P4)
Waiting time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)
Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (6 + 0 + 16 + 23)/4 = 45/4
= 11.25 milliseconds
Turn Around Time (TAT) for P1 = 16 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 6 ms
(Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (5 – 5) + 6 = 0 + 6)
Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)
Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P2 + P4 + P3 + P1)) / 4
= (16 + 6 + 23 + 28)/4 = 73/4
= 18.25 milliseconds

Priority based preemptive scheduling gives Real-Time attention to high priority tasks. Thus priority based preemptive scheduling is adopted in systems which demands 'Real-Time' behaviour. Most of the RTOSs make use of the preemptive priority based scheduling algorithm for process scheduling. Preemptive priority based scheduling also possesses the same drawback of non-preemptive priority based scheduling– '*Starvation*'. This can be eliminated by the '*Aging*' technique. Refer the section Non-preemptive priority based scheduling for more details on '*Starvation*' and '*Aging*'.

10.6 THREADS, PROCESSES AND SCHEDULING: PUTTING THEM ALTOGETHER

LO 6 Explain the different Inter Process Communication (IPC) mechanisms used by tasks/process to communicate and co-operate each other in a multitasking environment

So far we discussed about threads, processes and process/thread scheduling. Now let us have a look at how these entities are addressed in a real world implementation. Let's examine the following pieces of code.

```
//*********************************************************************
//Process 1
//*********************************************************************
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
//*********************************************************************
//Thread for executing Task
//*********************************************************************
void Task(void) {
while (1)
```

```
{
  //Perform some task
  //Task execution time is 7.5 units of execution
  //Sleep for 17.5 units of execution
  Sleep(17.5); //Parameter given is not in milliseconds
               //Repeat task
 }
}
//*********************************************************************
//Main Thread.
//*********************************************************************
void main(void) {
 DWORD id;
 HANDLE hThread;
 //Create thread with normal priority
 //*********************************************************************
 hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Task,
                        (LPVOID)0, 0, &id);
 if (NULL == hThread)
 {//Thread Creation failed. Exit process
  printf("Creating thread failed : Error Code =   %d", GetLastError());
  return;
 }
 WaitForSingleObject(hThread, INFINITE);
 return;
}
//*********************************************************************
//Process 2
//*********************************************************************
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
//*********************************************************************
//Thread for executing Task
//*********************************************************************
void Task(void) {
 while (1)
 {
  //Perform some task
  ///Task execution time is 10 units of execution
  //Sleep for 5 units of execution
  Sleep(5); //Parameter given is not in milliseconds
            //Repeat task
 }
}
//*********************************************************************
//Main Thread.
```

```
//*********************************************************************
void main(void) {
 DWORD id;
 HANDLE hThread;
 //Create thread with above normal priority
 //*********************************************************************
 hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Task,
                        (LPVOID)0, CREATE_SUSPENDED, &id);
 if (NULL == hThread)
 {//Thread Creation failed. Exit process
  printf("Creating thread failed : Error Code = %d", GetLastError());
  return;
 }
 SetThreadPriority(hThread, THREAD_PRIORITY_ABOVE_NORMAL);
 ResumeThread(hThread);
 WaitForSingleObject(hThread, INFINITE);
 return;
}
```

The first piece of code represents a process (Process 1) with priority normal and it performs a task which requires 7.5 units of execution time. After performing this task, the process sleeps for 17.5 units of execution time and this is repeated forever. The second piece of code represents a process (Process 2) with priority above normal and it performs a task which requires 10 units of execution time. After performing this task, the process sleeps for 5 units of execution time and this is repeated forever. Process 2 is of higher priority compared to process 1, since its priority is above 'Normal'.

Now let us examine what happens if these processes are executed on a Real-Time kernel with pre-emptive priority based scheduling policy. Imagine Process 1 and Process 2 are ready for execution. Both of them enters the 'Ready' queue and the scheduler picks up Process 2 for execution since it is of higher priority (Assuming there is no other process running/ready for execution, when both the processes are 'Ready' for execution) compared to Process 1. Process 2 starts executing and runs until it executes the Sleep instruction (i.e. after10 units of execution time). When the Sleep instruction is executed, Process 2 enters the wait state. Since Process 1 is waiting for its turn in the 'Ready' queue, the scheduler picks up it for execution, resulting in a context switch. The Process Control Block (PCB) of Process 2 is updated with the values of the Program Counter (PC), stack pointer, etc. at the time of context switch. The estimated task execution time for Process 1 is 7.5 units of execution time and the sleeping time for Process 2 is 5 units of execution. After 5 units of execution time, Process 2 enters the 'Ready' state and moves to the 'Ready' queue. Since it is of higher priority compared to the running process, the running process (Process 1) is pre-empted and Process 2 is scheduled for execution. Process 1 is moved to the 'Ready' queue, resulting in context switching. The Process Control Block of Process 1 is updated with the current values of the Program Counter (PC), Stack pointer, etc. when the context switch is happened. The Program Counter (PC), Stack pointer, etc. for Process 2 is loaded with the values stored in the Process Control Block (PCB) of Process 2 and Process 2 continues its execution form where it was stopped earlier. Process 2 executes the Sleep instruction after 10 units of execution time and enters the wait state. At this point Process 1 is waiting in the 'Ready' queue and it requires 2.5 units of execution time for completing the task associated with it (The total time for completing the task is 7.5 units of time, out of this it has already completed 5 units of execution when Process 2 was in the wait state). The scheduler schedules Process 1 for execution. The Program Counter (PC), Stack pointer, etc. for Process 1 is

loaded with the values stored in the Process Control Block (PCB) of Process 1 and Process 1 continues its execution form where it was stopped earlier. After 2.5 units of execution time, Process 1 executes the Sleep instruction and enters the wait state. Process 2 is already in the wait state and the scheduler finds no other process for scheduling. In order to keep the CPU always busy, the scheduler runs a dummy process (task) called '*IDLE PROCESS (TASK)*'. The '*IDLE PROCESS (TASK)*' executes some dummy task and keeps the CPU engaged. The execution diagram depicted in Fig. 10.14 explains the sequence of operations.

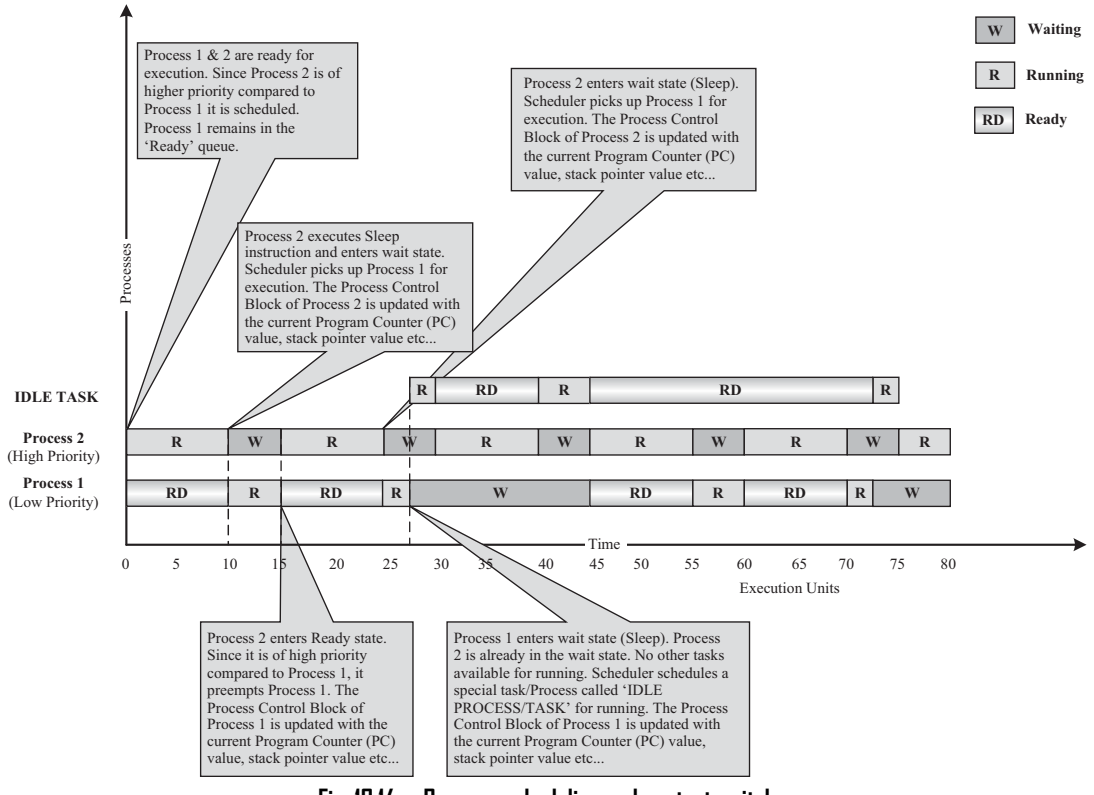


Fig. 10.14 Process scheduling and context switch

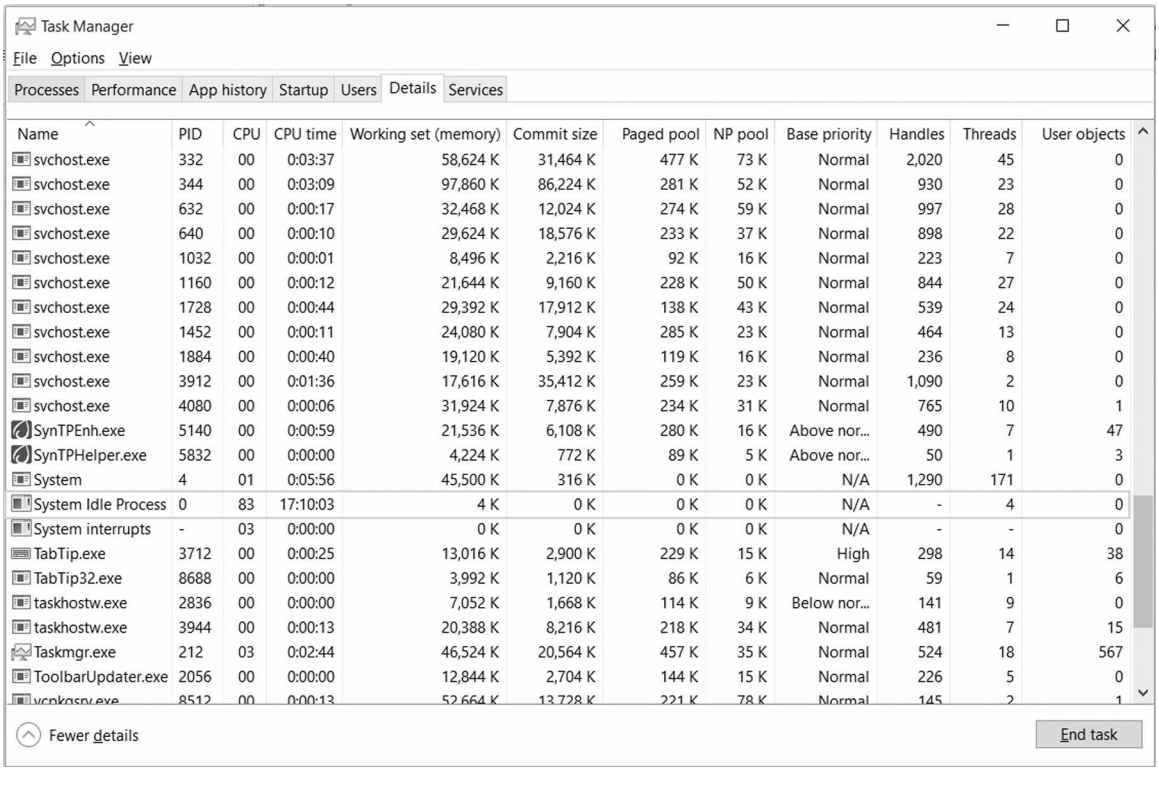
The implementation of the '*IDLE PROCESS (TASK)*' is dependent on the kernel and a typical implementation for a desktop OS may look like. It is simply an endless loop.

```
void Idle_Process (void)
{
//Simply wait.
//Do nothing...
while(1);
}
```

The Real-Time kernels deployed in embedded systems, where operating power is a big constraint (like systems which are battery powered); the '*IDLE TASK*' is used for putting the CPU into *IDLE* mode for saving the power. A typical example is the RTX51 Tiny Real-Time kernel, where the '*IDLE TASK*' sets the 8051 CPU to *IDLE* mode, a power saving mode. In the '*IDLE*' mode, the program execution is halted and

all peripherals and the interrupt system continues its operation. Once the CPU is put into the '*IDLE*' mode, it comes out of this mode when an Interrupt occurs or when the RTX51 Tiny Timer Tick Interrupt (The timer interrupt used for task scheduling in Round robin scheduling) occurs. It should be noted that the '*IDLE PROCESS (TASK)*' execution is not pre-emptive priority scheduling specific, it is applicable to all types of scheduling policies which demand 100% CPU utilisation/CPU power saving.

Back to the desktop OS environment, let's analyse the process, threads and scheduling in the Windows desktop environment. Windows provides a utility called task manager for monitoring the different process running on the system and the resources used by each process. A snapshot of the process details returned by the task manager for Windows 10 kernel is shown in Fig. 10.15. It should be noted that this snap- shot is purely machine dependent and it varies with the number of processes running on the machine. 'Name' represents the name of the process. 'PID' represents the Process Identification Number (Process ID). As mentioned in the 'Threads and Process' section, when a process is created an ID is associated to it. CPU usage gives the % of CPU utilised by the process during an interval. 'CPU Time' gives the total processor time, in seconds, used by a process since it started. 'Working set (memory)' represents the amount of memory in the private working set plus the amount of memory the process is using that can be shared by other processes. 'Commit Size' represents the amount of virtual memory that's reserved for use by a process. 'Paged Pool' represents the amount of pageable kernel memory allocated by the kernel or drivers on behalf of a process. Pageable memory is memory that can be written to another storage medium, such as the hard disk. 'NP Pool' is the amount of non-pageable kernel memory allocated by the kernel or drivers on behalf of a process.



Name	PID	CPU	CPU time	Working set (memory)	Commit size	Paged pool	NP pool	Base priority	Handles	Threads	User objects
svchost.exe	332	00	0:03:37	58,624 K	31,464 K	477 K	73 K	Normal	2,020	45	0
svchost.exe	344	00	0:03:09	97,860 K	86,224 K	281 K	52 K	Normal	930	23	0
svchost.exe	632	00	0:00:17	32,468 K	12,024 K	274 K	59 K	Normal	997	28	0
svchost.exe	640	00	0:00:10	29,624 K	18,576 K	233 K	37 K	Normal	898	22	0
svchost.exe	1032	00	0:00:01	8,496 K	2,216 K	92 K	16 K	Normal	223	7	0
svchost.exe	1160	00	0:00:12	21,644 K	9,160 K	228 K	50 K	Normal	844	27	0
svchost.exe	1728	00	0:00:44	29,392 K	17,912 K	138 K	43 K	Normal	539	24	0
svchost.exe	1452	00	0:00:11	24,080 K	7,904 K	285 K	23 K	Normal	464	13	0
svchost.exe	1884	00	0:00:40	19,120 K	5,392 K	119 K	16 K	Normal	236	8	0
svchost.exe	3912	00	0:01:36	17,616 K	35,412 K	259 K	23 K	Normal	1,090	2	0
svchost.exe	4080	00	0:00:06	31,924 K	7,876 K	234 K	31 K	Normal	765	10	1
SynTPEnh.exe	5140	00	0:00:59	21,536 K	6,108 K	280 K	16 K	Above nor...	490	7	47
SynTPHelper.exe	5832	00	0:00:00	4,224 K	772 K	89 K	5 K	Above nor...	50	1	3
System	4	01	0:05:56	45,500 K	316 K	0 K	0 K	N/A	1,290	171	0
System Idle Process	0	83	17:10:03	4 K	0 K	0 K	0 K	N/A	-	4	0
System interrupts	-	03	0:00:00	0 K	0 K	0 K	0 K	N/A	-	-	0
TabTip.exe	3712	00	0:00:25	13,016 K	2,900 K	229 K	15 K	High	298	14	38
TabTip32.exe	8688	00	0:00:00	3,992 K	1,120 K	86 K	6 K	Normal	59	1	6
taskhostw.exe	2836	00	0:00:00	7,052 K	1,668 K	114 K	9 K	Below nor...	141	9	0
taskhostw.exe	3944	00	0:00:13	20,388 K	8,216 K	218 K	34 K	Normal	481	7	15
Taskmgr.exe	212	03	0:02:44	46,524 K	20,564 K	457 K	35 K	Normal	524	18	567
ToolbarUpdater.exe	2056	00	0:00:00	12,844 K	2,704 K	144 K	15 K	Normal	226	5	0
vcpkgsrv.exe	8512	00	0:00:13	52,664 K	13,728 K	221 K	78 K	Normal	145	2	1

Fig. 10.15 Windows 10 Task Manager for monitoring process and resource usage

The non-paged memory cannot be swapped to the secondary storage disk. 'Base Priority' represents the priority of the process (A precedence ranking that determines the order in which the threads of a process are scheduled.). As mentioned in an earlier section, a process may contain multiple threads. The 'Threads' section gives the number of threads running in a process. 'Handles' reflects the number of object handles owned by the process. This value is the reflection of the object handles present in the process's object table. 'User Objects' reflects the number of objects active in the user mode for a process (A USER object is an object from Window Manager, which includes windows, menus, cursors, icons, hooks, accelerators, monitors, keyboard layouts, and other internal objects). Use 'Ctrl' + 'Alt' + 'Del' key for accessing the task manager and select the 'Details' tab. Right click the mouse on the row title header displaying the parameters and choose 'Select columns' option to select the different monitoring parameters for a process.

10.7 TASK COMMUNICATION

LO 7 Identify the RPC based Inter Process 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: In the co-operating interaction model 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.

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.

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 explained below.

10.7.1 Shared Memory

Processes share some area of the memory to communicate among them (Fig. 10.16). 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. It is same as the real world example where 'Notice Board' is used by corporate to publish the public information among the employees (The only exception is; only corporate have the right to modify the information published on the Notice board and employees are given 'Read' only access, meaning it is only a one way channel).

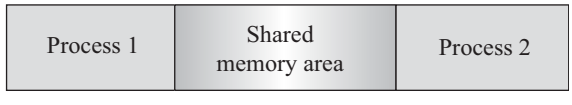


Fig. 10.16 Concept of Shared Memory

The implementation of shared memory concept is kernel dependent. Different mechanisms are adopted by different kernels for implementing this. A few among them are:

10.7.1.1 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 bi-directional information flow. 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 bi-directional pipe allows both reading and writing at one end. The unidirectional pipe can be visualised as

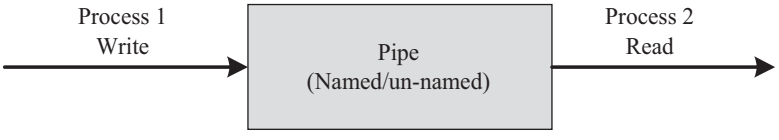


Fig. 10.17 Concept of Pipe for IPC

The implementation of 'Pipes' is also OS dependent. Microsoft® Windows Desktop Operating Systems support two types of 'Pipes' for Inter Process Communication. They are:

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. 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.

Please refer to the Online Learning Centre for details on the Pipe implementation under Windows Operating Systems.

Under VxWorks kernel, *pipe* is a special implementation of message queues. We will discuss the same in a latter chapter.

10.7.1.2 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 (of course certain synchronisation techniques should be applied to prevent inconsistent results). 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.

Windows Embedded Compact RTOS uses the memory mapped object based shared memory technique for Inter Process Communication (Fig. 10.18). The *CreateFileMapping (HANDLE hFile, LPSECURITY_ATTRIBUTES lpFileMappingAttributes, DWORD flProtect, DWORD dwMaximumSizeHigh,*

‡Client Server is a software architecture containing a client application and a server application. The application which sends request is known as client and the application which receives the request process it and sends a response back to the client is known as server. A server is capable of receiving request from multiple clients.

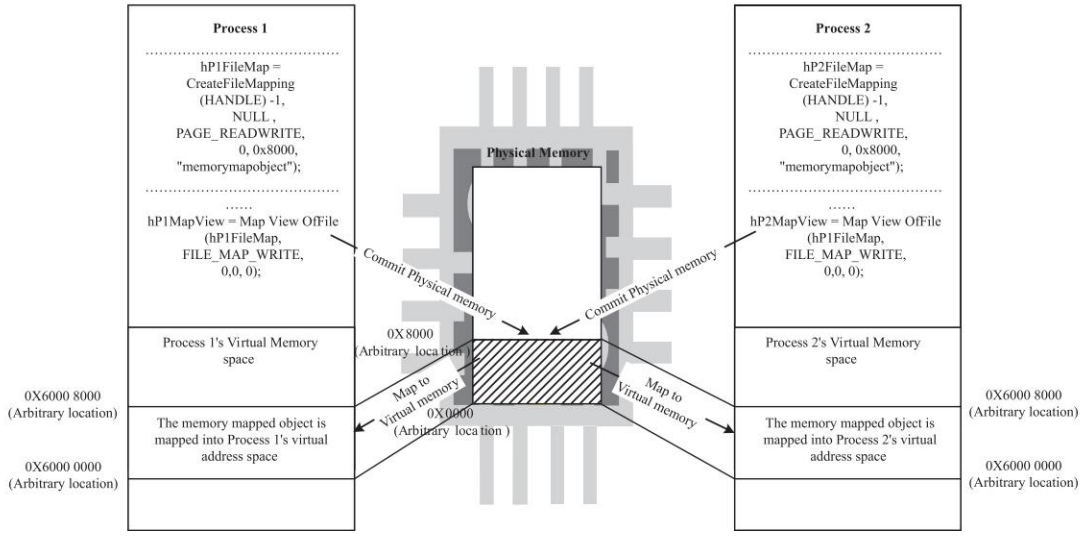


Fig. 10.18 Concept of memory mapped object

DWORD dwMaximumSizeLow, LPCTSTR lpName) system call is used for sharing the memory. This API call is used for creating a mapping from a file. In order to create the mapping from the system paging memory, the handle parameter should be passed as *INVALID_HANDLE_VALUE (–1)*. The *lpFileMappingAttributes* parameter represents the security attributes and it must be *NULL*. The *flProtect* parameter represents the read write access for the shared memory area. A value of *PAGE_READONLY* makes the shared memory read only whereas the value *PAGE_READWRITE* gives read-write access to the shared memory. The parameter *dwMaximumSizeHigh* specifies the higher order 32 bits of the maximum size of the memory mapped object and *dwMaximumSizeLow* specifies the lower order 32 bits of the maximum size of the memory mapped object. The parameter *lpName* points to a null terminated string specifying the name of the memory mapped object. The memory mapped object is created as unnamed object if the parameter *lpName* is *NULL*. If *lpName* specifies the name of an existing memory mapped object, the function returns the handle of the existing memory mapped object to the caller process. The memory mapped object can be shared between the processes by either passing the handle of the object or by passing its name. If the handle of the memory mapped object created by a process is passed to another process for shared access, there is a possibility of closing the handle by the process which created the handle while it is in use by another process. This will throw OS level exceptions. If the name of the memory object is passed for shared access among processes, processes can use this name for creating a shared memory object which will open the shared memory object already existing with the given name. The OS will maintain a usage count for the named object and it is incremented each time when a process creates/opens a memory mapped object with existing name. This will prevent the destruction of a shared memory object by one process while it is being accessed by another process. Hence passing the name of the memory mapped object is strongly recommended for memory mapped object based inter process communication. The *MapViewOfFile (HANDLE hFileMappingObject DWORD dwDesiredAccess, DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow, DWORD dwNumberOfBytesToMap)* system call maps a view of the memory mapped object to the address space of the calling process. The parameter *hFileMappingObject* specifies the handle to an existing memory mapped object. The *dwDesiredAccess* parameter represents the read write access for the mapped view area. A value of *FILE_MAP_WRITE* makes the view access read-write, provided the memory mapped object *hFileMappingObject* is created with read-write access, whereas

the value *FILE_MAP_READ* gives read only access to the shared memory, provided the memory mapped object *hFileMappingObject* is created with read-write/read only access. The parameter *dwFileOffsetHigh* specifies the higher order 32 bits and *dwFileOffsetLow* specifies the lower order 32 bits of the memory offset where mapping is to begin from the memory mapped object. A value of '0' for both of these maps the view from the beginning memory area of the memory object. *dwNumberOfBytesToMap* specifies the number of bytes of the memory object to map. If *dwNumberOfBytesToMap* is zero, the entire memory area owned by the memory mapped object is mapped. On successful execution, *MapViewOfFile* call returns the starting address of the mapped view. If the function fails it returns *NULL*. A mapped view of the memory mapped object is unmapped by the API call *UnmapViewOfFile (LPCVOID lpBaseAddress)*. The *lpBaseAddress* parameter specifies a pointer to the base address of the mapped view of a memory object that is to be unmapped. This value must be identical to the value returned by a previous call to the *MapViewOfFile* function. Calling *UnmapViewOfFile* cleans up the committed physical storage in a process's virtual address space. In other words, it frees the virtual address space of the mapping object. Under Windows NT Kernel, a process can open an existing memory mapped object by calling the API *OpenFileMapping(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName)*. The parameter *dwDesiredAccess* specifies the read write access permissions for the memory mapped object. A value of *FILE_MAP_ALL_ACCESS* provides read-write access, whereas the value *FILE_MAP_READ* allocates only read access and *FILE_MAP_WRITE* allocates write only access. The parameter *bInheritHandle* specifies the handle inheritance. If this parameter is *TRUE*, the calling process inherits the handle of the existing object, otherwise not. The parameter *lpName* specifies the name of the existing memory mapped object which needs to be opened. Windows CE 5.0 does not support handle inheritance and hence the API call *OpenFileMapping* is not supported.

The following sample code illustrates the creation and accessing of memory mapped objects across multiple processes. The first piece of code illustrates the creation of a memory mapped object with name "memorymappedobject" and prints the address of the memory location where the memory is mapped within the virtual address space of Process 1.

```
##include "stdafx.h"
#include <stdio.h>
#include <windows.h>
//*********************************************************************
//Process 1: Creates the memory mapped object and maps it to
//Process 1's Virtual Address space
//*********************************************************************
void main() {
 //Define the handle to Memory mapped Object
 HANDLE hFileMap;
 //Define the handle to the view of Memory mapped Object
 LPBYTE hMapView;
 printf("//***********************************************\n");
 printf(" Process 1\n");
 printf("//***********************************************\n");
 //Create an 8 KB memory mapped object
 hFileMap = CreateFileMapping((HANDLE)-1,
                 NULL, // default security attributes
                 PAGE_READWRITE, // Read-Write Access
                 0, //Higher order 32 bits of the memory mapping object
                 0x2000, //Lower order 32 bits of the memory mapping object
```

```
                TEXT("memorymappedobject")); // Memory mapped object name
 if (NULL == hFileMap)
 {
  printf("Memory mapped Object Creation Failed : Error Code : %d\n",
  GetLastError());
  //Memory mapped Object Creation failed. Return
  return;
 }
 //Map the memory mapped object to Process 1's address space
 hMapView = (LPBYTE)MapViewOfFile(hFileMap,
                 FILE_MAP_WRITE,
                 0, //Map the entire view
                 0,
                 0);
 if (NULL == hMapView)
 {
  printf("Mapping of Memory mapped view Failed : Error Code :%d\n",
  GetLastError());
  //Memory mapped view Creation failed. Return
  return;
 }
 else
 {
  //Successfully created the memory mapped view.
  //Print the start address of the mapped view
  printf("The memory is mapped to the virtual address starting at 0x%p\n",
 (void *)hMapView);
 }
 //Wait for user input to exit. Run Process 2 before providing
 //user input
 printf("Press any key to terminate Process 1");
 getchar();
 //Unmap the view
 UnmapViewOfFile(hMapView);
 //Close memory mapped object handle
 CloseHandle(hFileMap);
 return;
}
```

The piece of code given below corresponds to Process 2. It illustrates the accessing of an existing memory mapped object (memory mapped object with name "memorymappedobject" created by Process 1) and prints the address of the memory location where the memory is mapped within the virtual address space of Process 2. To demonstrate the application, the program corresponding to Process 1 should be executed first and the program corresponding to Process 2 should be executed following Process 1.

```
#include "stdafx.h"
#include <stdio.h>
```

```
#include <windows.h>
//*********************************************************************
//Process 2: Opens the memory mapped object created by Process 1
//Maps the object to Process 2's virtual address space.
//*********************************************************************
void main() {
 //Define the handle for the Memory mapped Object
 HANDLE hChildFileMap;
 //Define the handle for the view of Memory mapped Object
 LPBYTE hChildMapView;
 printf("//*********************************************\n");
 printf(" Process 2\n");
 printf("//*********************************************\n");
 //Create an 8 KB memory mapped object
 hChildFileMap = CreateFileMapping(INVALID_HANDLE_VALUE,
                    NULL, // default security attributes
                    PAGE_READWRITE, // Read-Write Access
                    0, //Higher order 32 bits of the memory mapping object
                    0x2000, //Lower order 32 bits of the memory mapping object
                    TEXT("memorymappedobject")); // Memory mapped object name
 if ((NULL == hChildFileMap) || (INVALID_HANDLE_VALUE == hChildFileMap))
 {
  printf("Memory mapped Object Creation Failed : Error Code : %d\n",
  GetLastError());
  //Memory mapped Object Creation failed. Return
  return;
 }
 else if (ERROR_ALREADY_EXISTS == GetLastError())
  {
   //A memory mapped object with given name exists already.
   printf("The named memory mapped object is already existing\n");
  }
 //Map the memory mapped object to Process 2's address space
 hChildMapView = (LPBYTE) MapViewOfFile(hChildFileMap,
                      FILE_MAP_WRITE, //Read-Write access
                      0,
                      //Map the entire view
                      0,
                      0);
 if (NULL == hChildMapView)
 {
  printf("Mapping of Memory mapped view Failed : Error Code : %d\n",
  GetLastError());
  //Memory mapped view Creation failed. Return
  return;
 }
```

```
else
 {
  //Successfully created the memory mapped view.
  //Print the start address of the mapped view
  printf("The memory mapped view starts at memory location 0x%p\n",
 (void *) hChildMapView);
 }
 //Wait for user input to exit.
 printf("Press any key to terminate Process 2");
 getchar();
 //Unmap the view
 UnmapViewOfFile(hChildMapView);
 //Close memory mapped object handle
 CloseHandle(hChildFileMap);
 return;
}
```

The output of the above programs when executed on Windows 10 OS in the sequence, Process 1 is executed first and Process 2 is executed while Process 1 waits for the user input from the keyboard, is given in Fig. 10.19.

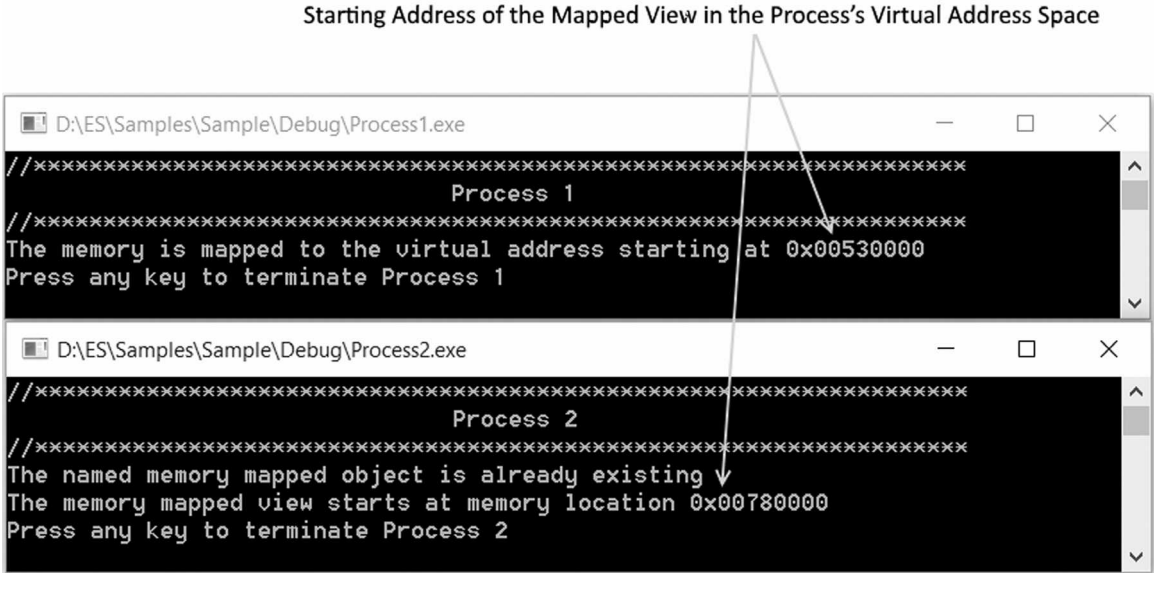


Fig. 10.19 Output of Win32 memory mapped object illustration program

The starting address of the memory mapped view in the virtual address space of the process is decided by the OS. However a process can request to map the view at a desired virtual address by using the *MapViewofFileEx* function. Depending on the availability of the address space the OS may grand the request or not. It is always safe to let the OS choose the address.

Reading and writing to a memory mapped area is same as any read write operation using pointers. The pointer returned by the API call *MapViewOfFile* can be used for this. The exercise of Read and Write operation is left to the readers. Proper care should be taken to avoid any conflicts that may arise due to the simultaneous read/write access of the shared memory area by multiple processes. This can be handled by applying various synchronisation techniques like events, mutex, semaphore, etc.

For using a memory mapped object across multiple threads of a process, it is not required for all the threads of the process to create/open the memory mapped object and map it to the thread's virtual address space. Since the thread's address space is part of the process's virtual address space, which contains the thread, only one thread, preferably the parent thread (main thread) is required to create the memory mapped object and map it to the process's virtual address space. The thread which creates the memory mapped object can share the pointer to the mapped memory area as global pointer and other threads of the process can use this pointer for reading and writing to the mapped memory area. If one thread of a process tries to create a memory mapped object with the same name as that of an existing mapping object, which is created by another thread of the same process, a new view of the mapping object is created at a different virtual address of the process. This is same as one process trying to create two views of the same memory mapped object☺.

10.7.2 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 info/data is passed through message passing. Also message passing is relatively fast and free from the synchronisation overheads compared to shared memory. Based on the message passing operation between the processes, message passing is classified into

10.7.2.1 Message Queue

Usually the process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called 'Message queue', which stores the messages temporarily in a system defined memory object, to pass it to the desired process (Fig. 10.20). Messages are sent and received through *send* (*Name of the process to which the message is to be sent, message*) and *receive* (*Name of the process from which the message is to be received, message*) methods. The messages are exchanged through a message queue. The implementation of the message queue, *send* and *receive* methods are OS kernel dependent. The Windows XP OS kernel maintains a single system message queue and one process/thread (Process and threads are used interchangeably here, since thread is the basic unit of process in windows) 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. For posting a message to a thread's message queue, the kernel fills a message structure *MSG* and copies it to the message queue of the thread. The message structure *MSG* contains the handle of the process/thread for which the message is intended, the message parameters, the time at which the message is posted, etc. A thread can simply post a message to another thread and can continue its operation or it may wait for a response from the thread to which the message is posted. 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, whereas 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.

The *PostMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)* or *PostThreadMessage (DWORD idThread, UINT Msg, WPARAM wParam, LPARAM lParam)* API is used by a thread in Windows for posting a message to its own message queue or to the message queue of another thread. *PostMessage* places a message at the end of a thread's message queue and returns immediately, without waiting for the thread to process the message. The function's parameters include a window handle, a message identifier, and two message parameters. The system copies these parameters to an *MSG* structure, and places the structure in the message queue. The *PostThreadMessage* is similar to *PostMessage*, except the first parameter is a thread identifier and it is used for posting a message to a specific thread message queue. The *SendMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)* API call sends a message to the thread specified by the handle *hWnd* and waits for the callee thread to process the message. The thread which calls the *SendMessage* API enters waiting state and waits for the message result from the thread to which the message is posted. The thread which invoked the *SendMessage* API call becomes blocked and the scheduler will not pick it up for scheduling.

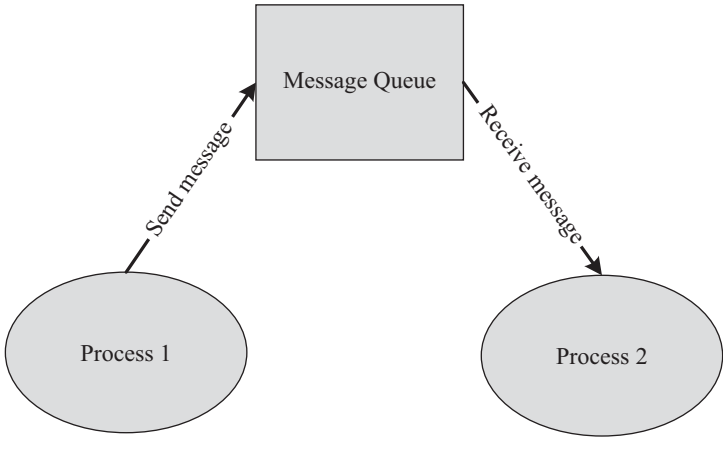


Fig. 10.20 Concept of message queue based indirect messaging for IPC

The Windows Embedded Compact operating system supports a special Point-to-Point Message queue implementation. The OS maintains a First In First Out (FIFO) buffer for storing the messages and each process can access this buffer for reading and writing messages. The OS also maintains a special queue, with single message storing capacity, for storing high priority messages (Alert messages). The creation and usage of message queues under Windows Embedded Compact OS is explained below.

The *CreateMsgQueue(LPCWSTR lpszName, LPMSGQUEUEOPTIONS lpOptions)* API call creates a message queue or opens a named message queue and returns a read only or write only handle to the message queue. A process can use this handle for reading or writing a message from/to of the message queue pointed by the handle. The parameter *lpszName* specifies the name of the message queue. If this parameter is *NULL*, an unnamed message queue is created. Processes can use the handle returned by the API call if the message queue is created without any name. If the message queue is created as named message queue, other processes can use the name of the message queue for opening the named message queue created by a process. Calling the *CreateMsgQueue* API with an existing named message queue as parameter returns a handle to the existing message queue. Under the Desktop Windows Operating Systems (Windows NT Kernel – Windows XP, Windows 8.1/10, etc.), each object type (viz. mutex, semaphores, events, memory maps, watchdog timers

and message queues) share the same namespace and the same name is not allowed for creating any of this. Windows CE/Embedded Compact kernel maintains separate namespace for each and supports the same name across different objects. The *lpOptions* parameter points to a *MSGQUEUEOPTIONS* structure that sets the properties of the message queue. The member details of the *MSGQUEUEOPTIONS* structure is explained below.

```
typedef MSGQUEUEOPTIONS_OS{
  DWORD dwSize;
  DWORD dwFlags;
  DWORD dwMaxMessages;
  DWORD cbMaxMessage;
  BOOL bReadAccess;
} MSGQUEUEOPTIONS, FAR* LPMSGQUEUEOPTIONS, *PMSGQUEUEOPTIONS;
```

The members of the structure are listed below.

Member	Description
dwSize	Specifies the size of the structure in bytes
dwFlags	Describes the behaviour of the message queue. Set to MSGQUEUE_NOPRECOMMIT to allocate message buffers on demand and to free the message buffers after they are read, or set to MSGQUEUE_ALLOW_BROKEN to enable a read or write operation to complete even if there is no corresponding writer or reader present.
dwMaxMessages	Specifies the maximum number of messages to queue at any point of time. Set this value to zero to specify no limit on the number of messages to queue at any point of time.
cbMaxMessage	Specifies the maximum number of bytes in each message. This value must be greater than zero.
bReadAccess	Specifies the Read Write access to the message queue. Set to TRUE to request read access to the queue. Set to FALSE to request write access to the queue.

On successful execution, the *CreateMsgQueue* API call returns a 'Read Only' or 'Write Only' handle to the specified queue based on the *bReadAccess* member of the *MSGQUEUEOPTIONS* structure *lpOptions*. If the queue with specified name already exists, a new handle, which points to the existing queue, is created and a following call to *GetLastError* returns *ERROR_ALREADY_EXISTS*. If the function fails it returns *NULL*. A single call to the *CreateMsgQueue* creates the queue for either 'read' or 'write' access. The *CreateMsgQueue* API should be called twice with the *bReadAccess* member of the *MSGQUEUEOPTIONS* structure *lpOptions* set to TRUE and FALSE respectively in successive calls for obtaining 'Read only' and 'Write only' handles to the specified message queue. The handle returning by *CreateMsgQueue* API call is an *event* handle and, if it is a 'Read Only' access handle, it is signalled by the message queue if a new message is placed in the queue. The signal is reset on reading the message by *ReadMsgQueue* API call. A 'Write Only' access handle to the message queue is signalled when the queue is no longer full, i.e. when there is room for accommodating new messages. Processes can monitor the handles with the help of the wait functions, viz. *WaitForSingleObject* or *WaitForMultipleObjects*. The *OpenMsgQueue(HANDLE hSrcProc, HANDLE hMsgQ, LPMSGQUEUEOPTIONS lpOptions)* API call opens an existing named or unnamed message queue. The parameter *hSrcProc* specifies the process handle of the process that owns the message queue and *hMsgQ* specifies the handle of the existing message queue (Handle to the message queue returned by the CreateMsgQueue function). As in the case of *CreateMsgQueue*, the *lpOptions* parameter points to a *MSGQUEUEOPTIONS* structure that sets the properties of the message queue. On successful execution the *OpenMsgQueue* API call returns a handle to the message queue and *NULL* if it fails. Normally

the *OpenMsgQueue* API is used for opening an unnamed message queue. The *WriteMsgQueue(HANDLE hMsgQ, LPVOID lpBuffer, DWORD cbDataSize, DWORD dwTimeout, DWORD dwFlags)* API call is used for writing a single message to the message queue pointed by the handle *hMsgQ*. *lpBuffer* points to a buffer that contains the message to be written to the message queue. The parameter *cbDataSize* specifies the number of bytes stored in the buffer pointed by *lpBuffer*, which forms a message. The parameter *dwTimeout* specifies the timeout interval in milliseconds for the message writing operation. A value of zero specifies the write operation to return immediately without blocking if the write operation cannot succeed. If the parameter is set to *INFINITE*, the write operation will block until it succeeds or the message queue signals the 'write only' handle indicating the availability of space for posting a message. The *dwFlags* parameter sets the priority of the message. If it is set to *MSGQUEUE_MSGALERT*, the message is posted to the queue as high priority or alert message. The Alert message is always placed in the front of the message queue. This function returns *TRUE* if it succeeds and *FALSE* otherwise.

The *ReadMsgQueue(HANDLE hMsgQ, LPVOID lpBuffer, DWORD cbBufferSize, LPDWORD lpNumberOfBytesRead, DWORD dwTimeout, DWORD* pdwFlags)* API reads a single message from the message queue. The parameter *hMsgQ* specifies a handle to the message queue from which the message needs to be read. *lpBuffer* points to a buffer for storing the message read from the queue. The parameter *cbBufferSize* specifies the size of the buffer pointed by *lpBuffer*, in bytes. *lpNumberOfBytesRead* specifies the number of bytes stored in the buffer. This is same as the number of bytes present in the message which is read from the message queue. *dwTimeout* specifies the timeout interval in milliseconds for the message reading operation. The timeout values and their meaning are same as that of the write message timeout parameter. The *dwFlags* parameter indicates the priority of the message. If the message read from the message queue is a high priority message (alert message), *dwFlags* is set to *MSGQUEUE_MSGALERT*. The function returns *TRUE* if it succeeds and *FALSE* otherwise. The *GetMsgQueueInfo (HANDLE hMsgQ, LPMSGQUEUEINFO lpInfo)* API call returns the information about a message queue specified by the handle *hMsgQ*. The message information is returned in a *MSGQUEUEINFO* structure pointed by *lpInfo*. The details of the *MSGQUEUEINFO* structure is explained below.

```
typedef MSGQUEUEINFO{
  DWORD dwSize;
  DWORD dwFlags;
  DWORD dwMaxMessages;
  DWORD cbMaxMessage;
  DWORD dwCurrentMessages;
  DWORD dwMaxQueueMessages;
  WORD wNumReaders;
  WORD wNumWriters;
} MSGQUEUEINFO, *PMSGQUEUEINFO, FAR* LPMSGQUEUEINFO;
```

The member variable details are listed below.

Member	Description
DwSize	Specifies the size of the buffer passed in.
dwFlags	Describes the behaviour of the message queue. It retrieves the *MSGQUEUEOPTIONS. dwFlags* passed when the message queue is created with *CreateMsgQueue* API call.
dwMaxMessages	Specifies the maximum number of messages to queue at any point of time. This reflects the *MSGQUEUEOPTIONS. dwMaxMessages* value passed when the message queue is created with *CreateMsgQueue* API call.

cbMaxMessage	Specifies the maximum number of bytes in each message. This reflects the *MSGQUEUEOPTIONS.cbMaxMessage* value passed when the message queue is created with *CreateMsgQueue* API call.
dwCurrentMessages	Specifies the number of messages currently existing in the specified message queue.
dwMaxQueueMessages	Specifies maximum number of messages that have ever been in the queue at one time.
wNumReaders	Specifies the number of readers (processes which opened the message queue for reading) subscribed to the message queue for reading.
wNumWriters	Specifies the number of writers (processes which opened the message queue for writing) subscribed to the message queue for writing.

The *GetMsgQueueInfo* API call returns *TRUE* if it succeeds and *FALSE* otherwise. The *CloseMsgQueue(HANDLE hMsgQ)* API call closes a message queue specified by the handle *hMsgQ*. If a process holds a 'read only' and 'write only' handle to the message queue, both should be closed for closing the message queue.

'Message queue' is the primary inter-task communication mechanism under VxWorks kernel. Message queues support two-way communication of messages of variable length. The two-way messaging between tasks can be implemented using one message queue for incoming messages and another one for outgoing messages. Messaging mechanism can be used for task-to task and task to Interrupt Service Routine (ISR) communication. We will discuss about the VxWorks' message queue implementation in a separate chapter.

10.7.2.2 Mailbox

Mailbox is an alternate form of 'Message queues' and it is used in certain Real-Time Operating Systems for IPC. Mailbox technique for IPC in RTOS is usually used 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. 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. The implementation of mailbox is OS kernel dependent. The MicroC/OS-II implements mailbox as a mechanism for inter-task communication. We will discuss about the mailbox based IPC implementation under MicroC/OS-II in a latter chapter. Figure 10.21 given below illustrates the mailbox based IPC technique.

10.7.2.3 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. The communication mechanisms used in RTX51 Tiny OS is an example for Signalling. The *os_send_signal* kernel call under RTX 51 sends a signal from one task to a specified task. Similarly the *os_wait* kernel call waits for a specified signal. Refer to the topic 'Round Robin Scheduling' under the section 'Priority based scheduling' for more details on Signalling in RTX51 Tiny OS. The VxWorks RTOS kernel also implements 'signals' for inter

process communication. Whenever a specified signal occurs it is handled in a signal handler associated with the signal. We will discuss about the signal based IPC mechanism for VxWorks' kernel in a later chapter.

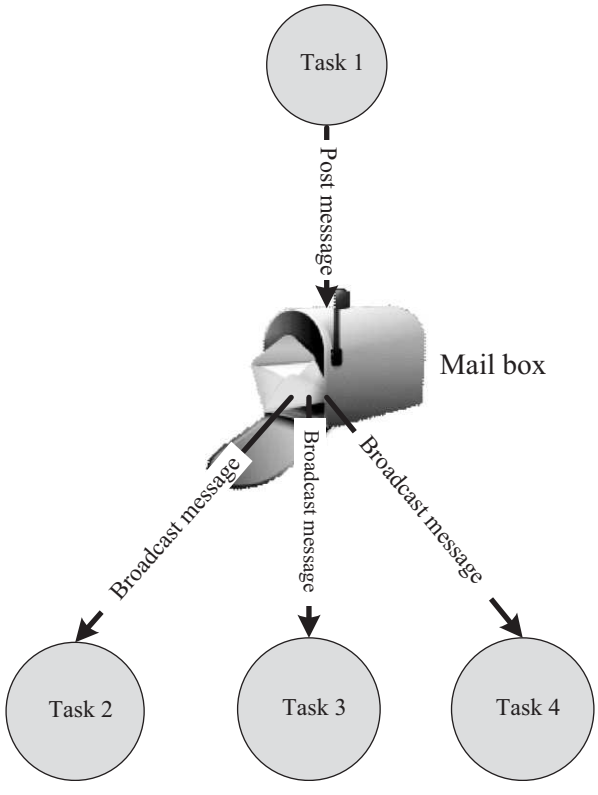
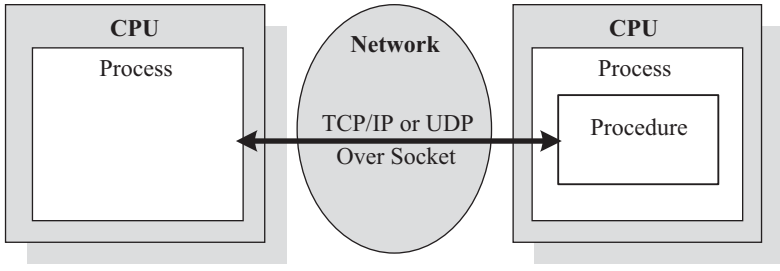


Fig. 10.21 Concept of Mailbox based indirect messaging for IPC

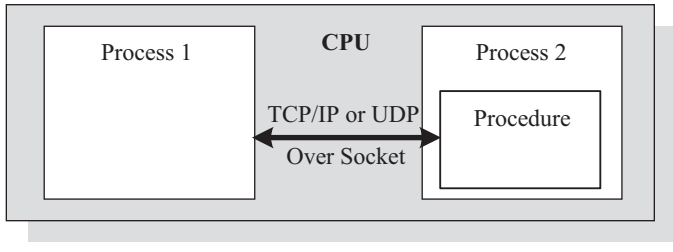
10.7.3 Remote Procedure Call (RPC) and Sockets

Remote Procedure Call or RPC (Fig. 10.22) 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.

It is possible to implement RPC communication with different invocation interfaces. In order to make the RPC communication compatible across all platforms it should stick on to certain standard formats. 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



Processes running on different CPUs which are networked



Processes running on same CPU

Fig. 10.22 Concept of Remote Procedure Call (RPC) for IPC

the execution of the procedure. The result from the remote procedure is returned back to the caller through mechanisms like callback functions.

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.

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. INET sockets are classified into:

1. Stream sockets
2. Datagram sockets

Stream sockets are connection oriented and they use TCP to establish a reliable connection. On the other hand, Datagram sockets rely on UDP for establishing a connection. The UDP connection is unreliable when compared to TCP. 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. 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. The following sample code illustrates the usage of socket for creating a client application under Windows OS. Winsock (Windows Socket 2) is the library implementing socket functions for Win32.

```
#include "stdafx.h"
#include <stdio.h>
#include <winsock2.h>
#include <Ws2tcpip.h>
//Specify the server address
#define SERVER "172.168.0.1"
//Specify the server port
#define PORT 5000
#pragma comment(lib, "Ws2_32.lib")
const int recvbuflen = 100;
char *sendbuf = "Hi from Client";
char recvbuffer[recvbuflen];

void main() {
 //*************************************************************
 // Initialise Winsock
 WSADATA wsaData;
 if (WSAStartup(MAKEWORD(2, 2), &wsaData) == NO_ERROR)
  printf("Winsock Initialisation succeeded...\n");
 else
 {
  printf("Winsock Initialisation failed...\n");
  return;
 }
 //*************************************************************
 // Create a SOCKET for connecting to server
 SOCKET MySocket;
 MySocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
 if (MySocket == INVALID_SOCKET)
 {
  printf("Socket Creation failed...\n");
  WSACleanup();
  return;
 }
 else
 {
  printf("Successfully created the socket...\n");
  //*************************************************************
  // Set the Socket type, IP address and port of the server
  sockaddr_in ServerParams;
  ServerParams.sin_family = AF_INET;
```

```
  //ServerParams.sin_addr.s_addr = inet_addr(SERVER);
  if (inet_pton(AF_INET, SERVER, &(ServerParams.sin_addr)) != 1)
  {
   printf("Converting IP Address failed...\n");
   WSACleanup();
   return;
  }
  ServerParams.sin_port = htons(PORT);
  //*********************************************************
  // Connect to server.
  if (connect(MySocket, (SOCKADDR*)& ServerParams, sizeof(ServerParams)) ==
  SOCKET_ERROR)
  {
   printf("Connecting to Server failed...Error code: %d\n", GetLastError());
   WSACleanup();
   return;
  }
  else
  {
   printf("Successfully Connected to the server...\n");
   //*********************************************************
   // Send command to server
   if (send(MySocket, sendbuf, (int)strlen(sendbuf), 0) == SOCKET_ERROR) {
    printf("Sending data to server failed...\n");
    closesocket(MySocket);
    WSACleanup();
    return;
   }
   else
   {
    printf("Successfully sent command to server...\n");
    //*********************************************************
    // Receive a data packet
    if (recv(MySocket, recvbuffer, recvbuflen, 0) > 0)
    printf("Successfully Received a packet...\n The received packet is %s\n",
     recvbuffer);
    else
      printf("No response from server...\n");
    //*********************************************************
    //Close Socket
    closesocket(MySocket);
    WSACleanup();
    return;
   }
  }
 }
 getchar();
}
```

The above application tries to connect to a server machine with IP address 172.168.0.1 and port number 5000. Change the values of *SERVER* and *PORT* to connect to a machine with different IP address and port number. If the connection is success, it sends the data "Hi from Client" to the server and waits for a response from the server and finally terminates the connection.

Under Windows, the socket function library *Winsock* should be initiated before using the socket related functions. The function *WSAStartup* performs this initiation. The *socket()* function call creates a socket. The socket type, connection type and protocols for communication are the parameters for socket creation. Here the socket type is *INET* (*AF_INET*) and connection type is stream socket (*SOCK_STREAM*). The protocol selected for communication is *TCP/IP* (*IPPROTO_TCP*). After creating the socket it is connected to a server. For connecting to server, the server address and port number should be indicated in the connection request. The *sockaddr_in* structure specifies the socket type, IP address and port of the server to be connected to. The *connect ()* function connects the socket with a specified server. If the server grants the connection request, the *connect ()* function returns success. The *send()* function is used for sending data to a server. It takes the socket name and data to be sent as parameters. Data from server is received using the function call *recv()*. It takes the socket name and details of buffer to hold the received data as parameters. The TCP/IP network stack expects network byte order (Big Endian: Higher order byte of the data is stored in lower memory address location) for data. The function *htons()* converts the byte order of an unsigned short integer to the network order. The *closesocket()* function closes the socket connection. On the server side, the server creates a socket using the function *socket()* and binds the socket with a port using the *bind()* function. It listens to the port bonded to the socket for any incoming connection request. The function *listen()* performs this. Upon receiving a connection request, the server accepts it. The function *accept()* performs the accepting operation. Now the connectivity is established. Server can receive and transmit data using the function calls *recv()* and *send()* respectively. The implementation of the server application is left to the readers as an exercise.

10.8 TASK SYNCHRONISATION

LO 8 State the need for task synchronisation in a multitasking environment

In a multitasking environment, multiple processes run concurrently (in pseudo parallelism) and share the system resources. Apart from this, each process has its own boundary wall and they 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. What could be the result in these scenarios? Obviously unexpected results. How these issues can be addressed? The solution is, make each process aware of the access of a shared resource either directly or indirectly. 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 synchronisation issues may arise in a multitasking environment if processes are not synchronised properly. The following sections describe the major task communication synchronisation issues observed in multitasking and the commonly adopted synchronisation techniques to overcome these issues.

10.8.1 Task Communication/Synchronisation Issues

10.8.1.1 Racing

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

```
#include "stdafx.h"
#include <windows.h>
```