SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT - I

RTOS PROGRAMMING-SECA5302

UNIT I - **INTRODUCTION AND INTERNALS**

Introduction to Real-Time Systems, Classification of real time systems, Difference between GPOS and RTOS- Real Time Kernels - RTOS Architecture- Features of RTOS- POSIX-RT standard

Introduction to Real-Time Systems

Real-time operating system (RTOS) is an operating system intended to serve real time application that process data as it comes in, mostly without buffer delay. The full form of RTOS is Real time operating system. In a RTOS, Processing time requirement are calculated in tenths of seconds increments of time. It is time-bound system that can be defined as fixed time constraints. In this type of system, processing must be done inside the specified constraints. Otherwise, the system will fail.

Usage of RTOS

Here are important reasons for using RTOS:

- **It offers priority-based scheduling, which allows you to separate analytical processing from non-critical processing.**
- **The Real time OS provides API functions that allow cleaner and smaller application code.**
- **Abstracting timing dependencies and the task-based design results in fewer interdependencies between modules.**
- **RTOS offers modular task-based development, which allows modular task-based testing.**
- **The task-based API encourages modular development as a task, will typically have a clearly defined role. It allows designers/teams to work independently on their parts of the project.**
- **An RTOS is event-driven with no time wastage on processing time for the event which is not occur**

Components of RTOS

Here, are important Component of RTOS

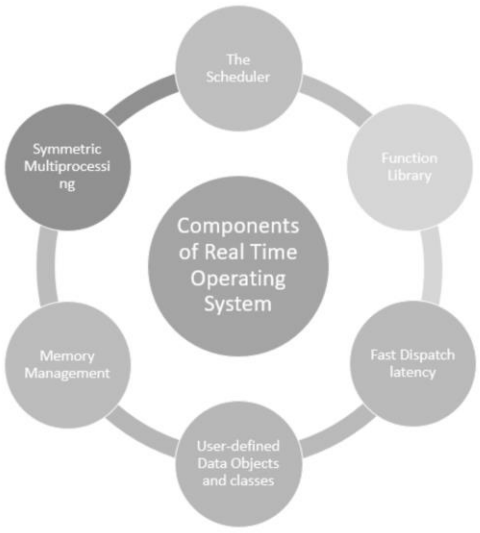


Fig:1.1: Components of RTOS

The Scheduler: This component of RTOS tells that in which order, the tasks can be executed which is generally based on the priority.

Symmetric Multiprocessing (SMP): **It is a number of multiple different tasks that can be handled by the RTOS so that parallel processing can be done.**

Function Library: It is an important element of RTOS that acts as an interface that helps you to connect kernel and application code. This application allows you to send the requests to the Kernel using a function library so that the application can give the desired results.

Memory Management: this element is needed in the system to allocate memory to every program, which is the most important element of the RTOS.

Fast dispatch latency: It is an interval between the termination of the task that can be identified by the OS and the actual time taken by the thread, which is in the ready queue, that has started processing.

User-defined data objects and classes: RTOS system makes use of programming languages like C or C++, which should be organized according to their operation.

Types of RTOS

Three types of RTOS systems are:

Hard Real Time :

In Hard RTOS, the deadline is handled very strictly which means that given task must start executing on specified scheduled time, and must be completed within the assigned time duration.

Example: Medical critical care system, Aircraft systems, etc.

Firm Real time:

These type of RTOS also need to follow the deadlines. However, missing a deadline may not have big impact but could cause undesired affects, like a huge reduction in quality of a product.

Example: Various types of Multimedia applications.

Soft Real Time:

Soft Real time RTOS, accepts some delays by the Operating system. In this type of RTOS, there is a deadline assigned for a specific job, but a delay for a small amount of time is acceptable. So, deadlines are handled softly by this type of RTOS.

Example: Online Transaction system and Livestock price quotation System.

Factors for selecting an RTOS

Here, are essential factors that you need to consider for selecting RTOS:

Performance: Performance is the most important factor required to be considered while selecting for a RTOS.

Middleware: if there is no middleware support in Real time operating system, then the issue of time-taken integration of processes occurs.

Error-free: RTOS systems are error-free. Therefore, there is no chance of getting an error while performing the task.

Embedded system usage: Programs of RTOS are of small size. So we widely use RTOS for embedded systems.

Maximum Consumption: we can achieve maximum Consumption with the help of RTOS.

Task shifting: Shifting time of the tasks is very less.

Unique features: A good RTS should be capable, and it has some extra features like how it operates to execute a command, efficient protection of the memory of the system, etc.

24/7 performance: RTOS is ideal for those applications which require to run 24/7.

Here are important differences between GPOS and RTOS:

Table:1.1:Difference between in GPOS and RTOS

General-Purpose Operating System (GPOS)	Real-Time Operating System (RTOS)
It used for desktop PC and laptop	It is only applied to the embedded application.
Process-based Scheduling	Time-based scheduling used like round-robin scheduling
Interrupt latency is not considered as important as in RTOS	Interrupt lag is minimal, which is measured in a few microseconds.
No priority inversion mechanism is present in the system.	The priority inversion mechanism is current. So it cannot modify by the system.
Kernel's operation may or may not be preempted.	Kernel's operation can be preempted.
Priority inversion remain unnoticed	No predictability guarantees

Applications of Real Time Operating System

Real-time systems are used in:

- **Airlines reservation system.**
- **Air traffic control system.**
- **Systems that provide immediate updating.**
- **Used in any system that provides up to date and minute information on stock prices.**
- **Defense application systems like RADAR.**
- **Networked Multimedia Systems**
- **Command Control Systems**
- **Internet Telephony**
- **Anti-lock Brake Systems**
- **Heart Pacemaker**

Disadvantages of RTOS

Here, are drawbacks/cons of using RTOS system:

- **RTOS system can run minimal tasks together, and it concentrates only on those applications which contain an error so that it can avoid them.**
- **RTOS is the system that concentrates on a few tasks. Therefore, it is really hard for these systems to do multi-tasking.**
- **Specific drivers are required for the RTOS so that it can offer fast response time to interrupt signals, which helps to maintain its speed.**
- **Plenty of resources are used by RTOS, which makes this system expensive.**
- **The tasks which have a low priority need to wait for a long time as the RTOS maintains the accuracy of the program, which are under execution.**
- **Minimum switching of tasks is done in Real time operating systems.**
- **It uses complex algorithms which is difficult to understand.**
- **RTOS uses lot of resources, which sometimes not suitable for the system.**

RTOS Architecture – Kernel

RTOS Architecture

For simpler applications, RTOS is usually a kernel but as complexity increases, various modules like networking protocol stacks debugging facilities, device I/Os are includes in addition to the kernel. The general architecture of RTOS is shown in the below fig 1.2

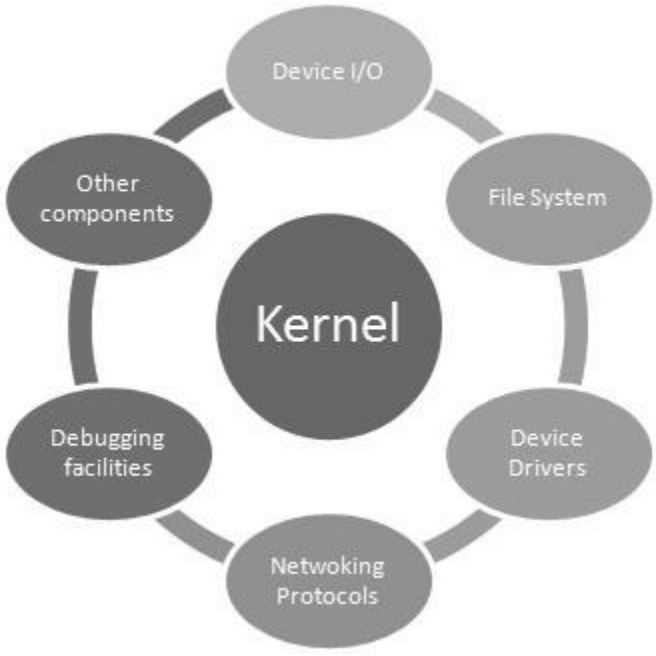


Fig:1.2: **RTOS Architecture**

Kernel

RTOS kernel acts as an abstraction layer between the hardware and the applications. There are three broad categories of kernels

- **Monolithic kernel**

Monolithic kernels are part of Unix-like operating systems like Linux, FreeBSD etc. A monolithic kernel is one single program that contains all of the code necessary to perform every kernel related task. It runs all basic system services (i.e. process and memory management, interrupt handling and I/O communication, file system, etc) and provides powerful abstractions of the underlying hardware. Amount of context switches and messaging involved are greatly reduced which makes it run faster than microkernel.

- **Microkernel**

It runs only basic process communication (messaging) and I/O control. It normally provides only the minimal services such as managing memory protection, Inter process communication and the process management. The other functions such as running the hardware processes are not handled directly by micro kernels. Thus, micro kernels provide a smaller set of simple hardware abstractions. It is more stable than monolithic as the kernel is unaffected even if the servers failed (i.e., File System). Micro kernels are part of the operating systems like AIX, BeOS, Mach, Mac OS X, MINIX, and QNX. Etc

- **Hybrid Kernel**

Hybrid kernels are extensions of micro kernels with some properties of monolithic kernels. Hybrid kernels are similar to micro kernels, except that they include additional code in kernel space so that such code can run more swiftly than it would were it in user space. These are part of the operating systems such as Microsoft Windows NT, 2000 and XP. Dragon Fly BSD, etc

- **Exokernel**

Exokernels provides efficient control over hardware. It runs only services protecting the resources (i.e. tracking the ownership, guarding the usage, revoking access to resources, etc) by providing low-level interface for library operating systems and leaving the management to the application.

Six types of common services are shown in the following figure below and explained in subsequent sections

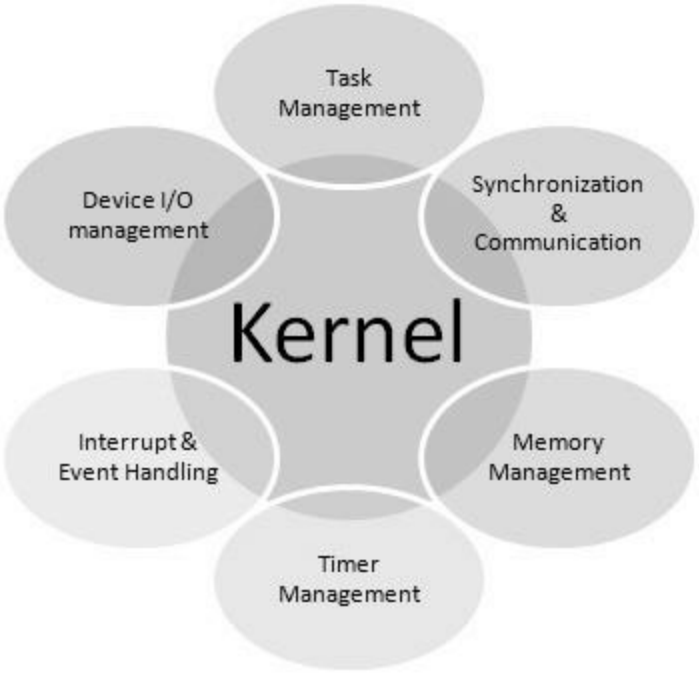


Fig:1.3: **Representation of Common Services Offered By a RTOS System**

Architecture – Task Management

Task Management

In RTOS, The application is decomposed into small, schedulable, and sequential program units known as "Task", a basic unit of execution and is governed by three time-critical properties; release time, deadline and execution time. Release time refers to the point in time from which the task can be executed. Deadline is the point in time by which the task must complete. Execution time denotes the time the task takes to execute.

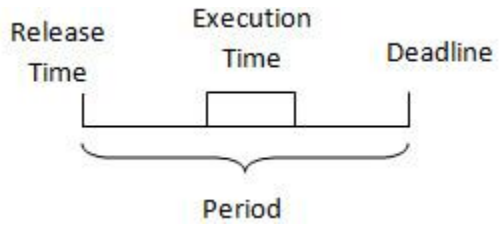


Fig:1.4: Use of RTOS for Time Management Application

Each task may exist in following states

Dormant : Task doesn't require computer time

Ready: Task is ready to go active state, waiting processor time

Active: Task is running

Suspended: Task put on hold temporarily

Pending: Task waiting for resource.

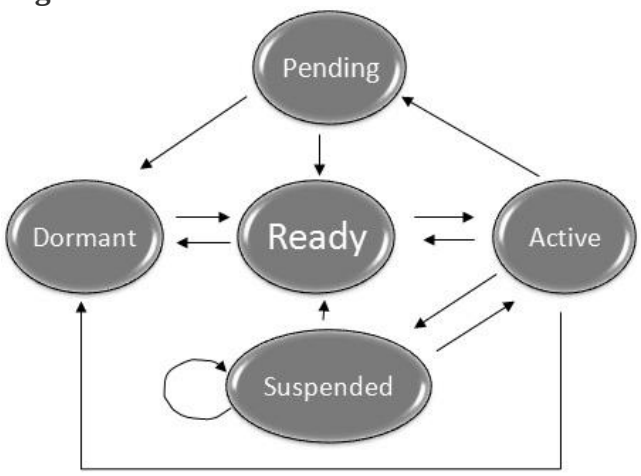


Fig:1.5: **Representation of Different Time Management Tasks Done by an RTOS**

During the execution of an application program, individual tasks are continuously changing from one state to another. However, only one task is in the running mode (i.e. given CPU control) at any point of the execution. In the process where CPU control is change from one task to another, context of the to-be-suspended task will be saved while context of the to-be-executed task will be retrieved, the process referred to as context switching. A task object is defined by the following set of components:

Task Control block: Task uses TCBs to remember its context. TCBs are data structures residing in RAM, accessible only by RTOS

Task Stack: These reside in RAM, accessible by stack pointer.

Task Routine: Program code residing in ROM

Scheduler: The scheduler keeps record of the state of each task and selects from among them that are ready to execute and allocates the CPU to one of them. Various scheduling algorithms are used in RTOS.

Polled Loop: Sequentially determines if specific task requires time.

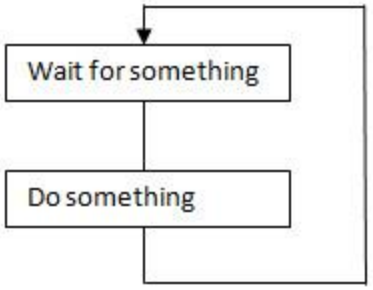


Fig:1.6: Process Flow of a Scheduler

Polled System with interrupts. In addition to polling, it takes care of critical tasks.

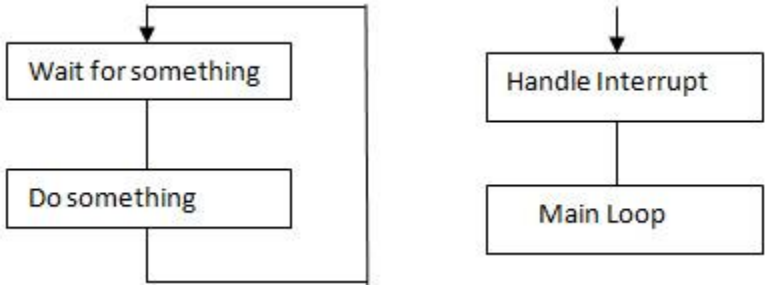


Fig:1.7: A Figure Illustrating Polled Systems with Interrupts

Round Robin : Sequences from task to task, each task getting a slice of time

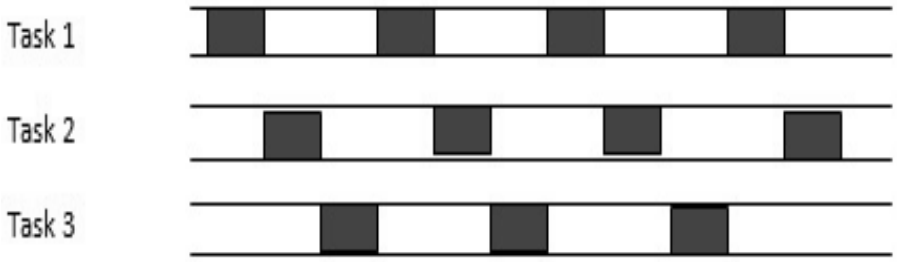


Fig:1.8: Round Robin Sequences From Task to Task

Hybrid System: Sensitive to sensitive interrupts, with Round Robin system working in background

Interrupt Driven: System continuously wait for the interrupts

Non pre-emptive scheduling or Cooperative Multitasking: Highest priority task executes for some time, then relinquishes control, re-enters ready state

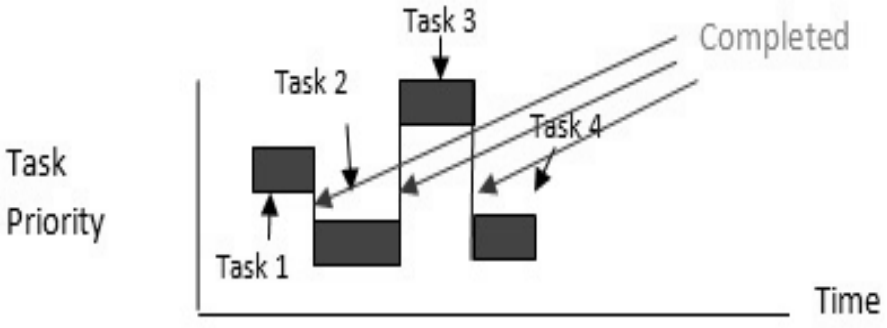


Fig:1.9: Non-Preemptive Scheduling or Cooperative Multitasking

Preemptive scheduling Priority multitasking: Current task is immediately suspended Control is given to the task of the highest priority at all time.

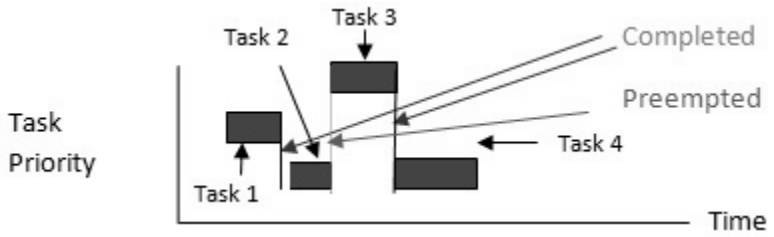


Fig:1.10: Preemptive Scheduling or Priority Multitasking

Dispatcher : The dispatcher gives control of the CPU to the task selected by the scheduler by performing context switching and changes the flow of execution.

Synchronization and communication:

Task Synchronization & inter task communication serves to pass information amongst tasks.

Task Synchronization

Synchronization is essential for tasks to share mutually exclusive resources (devices, buffers, etc) and/or allow multiple concurrent tasks to be executed (e.g. Task A needs a result from task B, so task A can only run till task B produces it.
Task synchronization is achieved using two types of mechanisms:

Event Objects

Event objects are used when task synchronization is required without resource sharing. They allow one or more tasks to keep waiting for a specified event to occur. Event object can exist either in triggered or non-triggered state. Triggered state indicates resumption of the task.

Semaphores.

A semaphore has an associated resource count and a wait queue. The resource count indicates availability of resource. The wait queue manages the tasks waiting for resources from the semaphore. A semaphore functions like a key that define whether a task has the access to the resource. A task gets an access to the resource when it acquires the semaphore.

There are three types of semaphore:

- **Binary Semaphores**
- **Counting Semaphores**
- **Mutually Exclusion (Mutex) Semaphores**

Semaphore functionality (Mutex) represented pictorially in the following figure

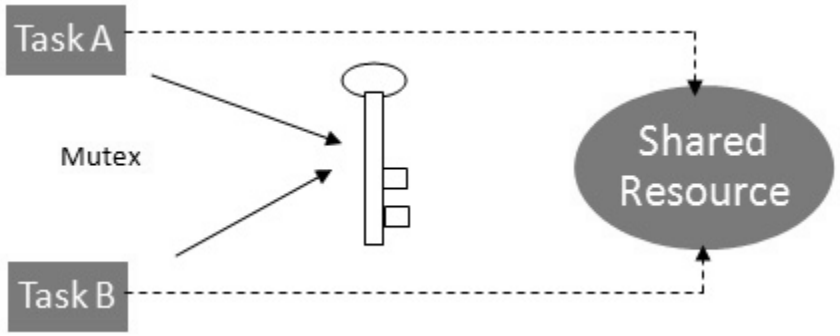


Fig:1.11*:* Architecture of Semaphore Functionality

Inter task communication

Inter task communication involves sharing of data among tasks through sharing of memory space, transmission of data, etc. Inter task communications is executed using following mechanisms

Message queues

A message queue is an object used for inter task communication through which task send or receive messages placed in a shared memory. The queue may follow 1) First In First Out (FIFO), 2) Last in First Out(LIFO) or 3) Priority (PRI) sequence. Usually, a message queue comprises of an associated queue control block (QCB), name, unique ID,

memory buffers, queue length, maximum message length and one or more task waiting lists. A message queue with a length of 1 is commonly known as a mailbox.

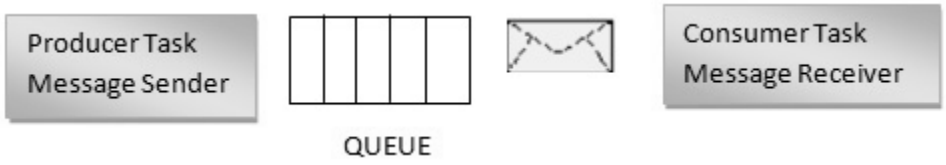


Fig:1.12: Flow of a Message Queue in a Mailbox

Pipes

A pipe is an object that provide simple communication channel used for unstructured data exchange among tasks. A pipe does not store multiple messages but stream of bytes. Also, data flow from a pipe cannot be prioritized.

Remote procedure call (RPC)

It permits distributed computing where task can invoke the execution of another task on a remote computer.

Memory Management

Two types of memory managements are provided in RTOS – Stack and Heap. Stack management is used during context switching for TCBs. Memory other than memory used for program code, program data and system stack is called heap memory and it is used for dynamic allocation of data space for tasks. Management of this memory is called heap management.

Timer Management

Tasks need to be performed after scheduled durations. To keep track of the delays, timers- relative and absolute are provided in RTOS.

Interrupt and event handling

RTOS provides various functions for interrupt and event handling, viz., Defining interrupt handler, creation and deletion of ISR, referencing the state of an ISR, enabling and disabling of an interrupt, etc. It also restricts interrupts from occurring when modifying a data structure, minimize interrupt latencies due to disabling of interrupts when RTOS is performing critical operations, minimizes interrupt response times.

Device I/O Management

RTOS generally provides large number of APIs to support diverse hardware device drivers.

Features of RTOS

Here are important features of RTOS:

- **Occupy very less memory**
- **Consume fewer resources**
- **Response times are highly predictable**
- **Unpredictable environment**
- **The Kernel saves the state of the interrupted task ad then determines which task it should run next.**
- **The Kernel restores the state of the task and passes control of the CPU for that task.**

POSIX (Portable Operating System Interface)

POSIX (Portable Operating System Interface) is a set of standard operating system interfaces based on the Unix operating system. The need for standardization arose because enterprises using computers wanted to be able to develop programs that could be moved among different manufacturer's computer systems without having to be recoded. Unix was selected as the basis for a standard system interface partly because it was "manufacturer-neutral." However, several major versions of Unix existed so there was a need to develop a common denominator system.

Informally, each standard in the POSIX set is defined by a decimal following the POSIX. Thus, POSIX.1 is the standard for an application program interface in the C language. POSIX.2 is the standard shell and utility interface (that is to say, the user's command interface with the operating system). These are the main two interfaces, but additional interfaces, such as POSIX.4 for thread management, have been developed or are being developed. The POSIX interfaces were developed under the auspices of the Institute of Electrical and Electronics Engineers (IEEE).

POSIX.1 and POSIX.2 interfaces are included in a somewhat larger interface known as the X/Open Programming Guide (also known as the "Single UNIX Specification" and "UNIX 03"). The Open Group, an industry standards group, owns the UNIX trademark and can thus "brand" operating systems that conform to the interface as "UNIX" systems. IBM's OS/390 is an example of an operating system that includes a branded UNIX interface.

Terms used in RTOS

Here, are essential terms used in RTOS:

- **Task – A set of related tasks that are jointly able to provide some system functionality.**

- **Job – A job is a small piece of work that can be assigned to a processor, and that may or may not require resources.**
- **Release time of a job – It's a time of a job at which job becomes ready for execution.**
- **Execution time of a job: It is time taken by job to finish its execution.**
- **Deadline of a job: It's time by which a job should finish its execution.**
- **Processors: They are also known as active resources. They are important for the execution of a job.**
- **Maximum it is the allowable response time of a job is called its relative deadline**
- **Response time of a job: It is a length of time from the release time of a job when the instant finishes.**
- **Absolute deadline: This is the relative deadline, which also includes its release time.**

Summary

RTOS is an operating system intended to serve real time application that process data as it comes in, mostly without buffer delay. It offers priority-based scheduling, which allows you to separate analytical processing from non-critical processing. Important components of RTOS system are:

- **The Scheduler**
- **Symmetric Multiprocessing**
- **Function Library**
- **Memory Management**
- **Fast dispatch latency and**
- **User-defined data objects and classes.**

RTOS system occupy very less memory and consume fewer resources. Performance is the most important factor required to be considered while selecting for a RTOS. General-Purpose Operating System (GPOS) is used for desktop PC and laptop while Real-Time Operating System (RTOS) only applied to the embedded application. Real-time systems are used in Airlines reservation system, Air traffic control system, etc. The biggest drawback of RTOS is that the system only concentrates on a few tasks.

TEXT / REFERENCE BOOKS

1. Jane W. S Liu, "Real Time Systems" Pearson Higher Education, 3rd Edition, 2000.
2. Raj Kamal, "Embedded Systems- Architecture, Programming and Design" Tata McGraw Hill, 2nd Edition, 2014.
3. Jean J. Labrosse, "Micro C/OS-I : The real time kernel" CMP Books, 2nd Edition,2015.
5. Richard Barry, "Mastering the Free RTOS: Real Time Kernel", Real Time Engineers Ltd, 1st Edition, 2016.
6. David E. Simon, " An Embedded Software Primer", Pearson Education Asia Publication

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT - II

RTOS PROGRAMMING-SECA5302

II PERFORMANCE METRICS AND SCHEDULING ALGORITHMS

Performance Metrics of RTOS- Task Specifications-Task state - Real Time Scheduling algorithms: Cyclic executive, Rate monotonic, IRIS and Least laxity scheduling- Schedulability Analysis

Performance Metrics of RTOS

An embedded system typically has enough CPU power to do the job, but typically only just enough — there is no excess. Memory size is usually limited. It is not unreasonably small, but there isn't likely to be any possibility of adding more. Power consumption is usually an issue and the software — its size and efficiency – can have a significant bearing on the number of watts burned by the embedded device. It is clear that it is vital in an embedded system that the real time operating system (RTOS) has the smallest possible impact on memory footprint and makes very efficient use of the CPU.

RTOS metrics

There are three areas of interest when looking at the performance and usage characteristics of an RTOS:

- **Memory – how much ROM and RAM does the kernel need and how is this affected by options and configuration**
- **Latency - which is broadly the delay between something happening and the response to that occurrence. This is a particular minefield of terminology and misinformation, but there are two essential latencies to consider: interrupt response and task scheduling.**
- **Performance of kernel services - How long does it take to perform specific actions. A key problem is that there is no real standardization. One possibility would be the Embedded Microprocessor Benchmark Consortium, but that does not cover all MCUs and, anyway, it is more oriented towards CPU, rather than MPU, benchmarking.**

RTOS Metrics – Memory footprint

As all embedded systems have some limitations on available memory, the requirements of an RTOS, on a given CPU, need to be understood. An OS will use both ROM and RAM. ROM, which is normally flash memory, is used for the kernel, along with code for the runtime library and any middleware components. This code – or parts of it – may be copied to RAM on boot up, as this can offer improved performance. There is also likely to be some read only data. If the kernel is statically configured, this data will include extensive

information about kernel objects. However, nowadays, most kernels are dynamically configured.

RAM space will be used for kernel data structures, including some or all of the kernel object information, again depending upon whether the kernel is statically or dynamically configured. There will also be some global variables. If code is copied from flash to RAM, that space must also be accounted for. There are a number of factors that affect the memory footprint of an RTOS. The CPU architecture is key. The number of instructions can vary drastically from one processor to another, so looking at size figures for, say, PowerPC gives no indication of what the ARM version might be like.

Embedded compilers generally have a large number of optimization settings. These can be used to reduce code size, but that will most likely affect performance. Optimizations affect ROM footprint, and also RAM. Data size can also be affected by optimization, as data structures can be packed or unpacked. Again both ROM and RAM can be affected. Packing data has an adverse effect on performance. Most RTOS products have a number of optional components. Obviously, the choice of those components will have a very significant effect upon memory footprint. Most RTOS kernels are scalable, which means that, all being well, only the code to support required functionality is included in the memory image. For some RTOS, scalability only applies to the kernel. For others, scalability is extended to the rest of the middleware.

Although an RTOS vendor may provide or publish memory usage information, you may wish to make measurements yourself in order to ensure that they are representative of the type of application that you are designing. These measurements are not difficult. Normally the map file, generated by the linker, gives the necessary memory utilization data. Different linkers produce different kinds of map files with varying amounts of information included in a variety of formats. Possibilities extend from a mass of hex numbers through to an interactive HTML document and everything in between. There are some specialized tools that extract memory usage information from executable files. An example is obj dump .

Interrupt latency

The time related performance measurements are probably of most concern to developers using an RTOS. A key characteristic of a real time system is its timely response to external events and an embedded system is typically notified of an event by means of an interrupt, so interrupt latency is critical.

- **System: the total delay between the interrupt signal being asserted and the start of the interrupt service routine execution.**
- **OS: the time between the CPU interrupt sequence starting and the**

initiation of the ISR. This is really the operating system overhead, but many people refer to it as the latency. This means that some vendors claim zero interrupt latency.

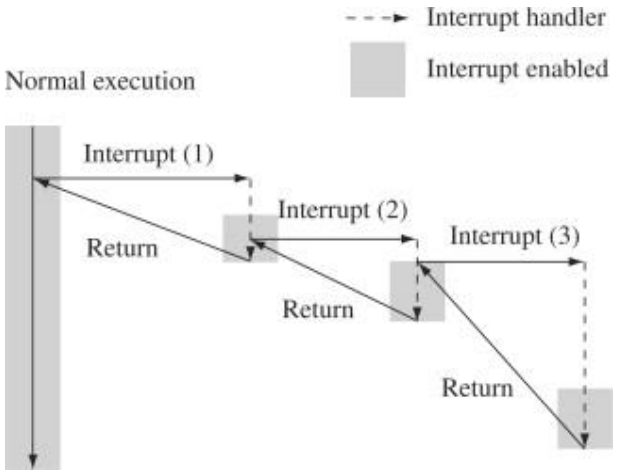


Fig:2.1: Interrupt latency

Measurement

Interrupt response is the sum of two distinct times:

$$T_{IL} = T_H + T_{OS}$$

where:

- T_H **is the hardware dependent time, which depends on the interrupt controller on the board as well as the type of the interrupt**
- T_{OS} **is the OS induced overhead**

Ideally, quoted figures should include the best and worst case scenarios. The worst case is when the kernel disables interrupts. To measure a time interval, like interrupt latency, with any accuracy, requires a suitable instrument. The best tool to use is an oscilloscope. One approach is to use one pin on a GPIO interface to generate the interrupt. This pin can be monitored on the 'scope. At the start of the interrupt service routine, another pin, which is also being monitored, is toggled. The interval between the two signals may be easily read from the instrument.

Importance

Many embedded systems are real time and it is those applications, along with fault tolerant systems, where knowledge of interrupt latency is important. If the requirement is to maximize bandwidth on a particular interface, the latency on that specific interrupt needs to be measured. To give an idea of numbers, the majority of systems exhibit no problems, even if they are subjected to interrupt latencies of tens of microseconds

Interrupt latency is the sum of the hardware dependent time, which depends on the interrupt controller as well as the type of the interrupt, and the OS induced overhead. Ideally, quoted figures should include the best and worst case scenarios. The worst case is when the kernel disables interrupts. To measure a time interval, like interrupt latency, with any accuracy, requires a suitable instrument and the best tool to use is an oscilloscope. One approach is to use one pin on a GPIO interface to generate the interrupt and monitor it on the 'scope. At the start of the interrupt service routine, another pin, which is also being monitored, is toggled and the interval between the two signals may be easily read.

Scheduling latency

A key part of the functionality of an RTOS is its ability to support a multi-threading execution environment. Being real time, the efficiency at which threads or tasks are scheduled is of some importance and the scheduler is at the core of an RTOS. so it is reasonable that a user might be interested in its performance. It is hard to get a clear picture of performance, as there is a wide variation in the techniques employed to make measurements and in the interpretation of the results.

There are really two separate measurements to consider:

- **The context switch time**
- **The time overhead that the RTOS introduces when scheduling a task**

Timing kernel services

An RTOS is likely to have a great many system service API (application program interface) calls, probably numbering into the hundreds. To assess timing, it is not useful to try to analyze the timing of every single call. It makes more sense to focus on the frequently used services.

For most RTOS there are four key categories of service call:

- **Threading services**
- **Synchronization services**

- **Inter-process communication services**
- **Memory services**

All RTOS vendors provide performance data for their products, some of which is more comprehensive than others. This information may be very useful, but can also be misleading if interpreted incorrectly. It is important to understand the techniques used to make measurements and the terminology used to describe the results. There are also trade-offs – generally size against speed – and these, too, need to be thoroughly understood. Without this understanding, a fair comparison is not possible. If timing is critical to your application, it is strongly recommend that you perform your own measurements. This enables you to be sure that the hardware and software environment is correct and that the figures are directly relevant to your application.

Real Time Scheduling algorithms

In real time operating systems(RTOS) most of the tasks are periodic in nature. Periodic data mostly comes from sensors, servo control, and real-time monitoring systems. In real time operating systems, these periodic tasks utilize most of the processor computation power. A real-time control system consists of many concurrent periodic tasks having individual timing constraints. These timing constraints include release time (ri), worst case execution time(Ci), period (ti) and deadline(Di) for each individual task Ti. Real time embedded systems have time constraints linked with the output of the system. The scheduling algorithms are used to determine which task is going to execute when more than one task is available in the ready queue.

The operating system must guarantee that each task is activated at its proper rate and meets its deadline. To ensure this, some periodic scheduling algorithms are used. There are basic two types of scheduling algorithms

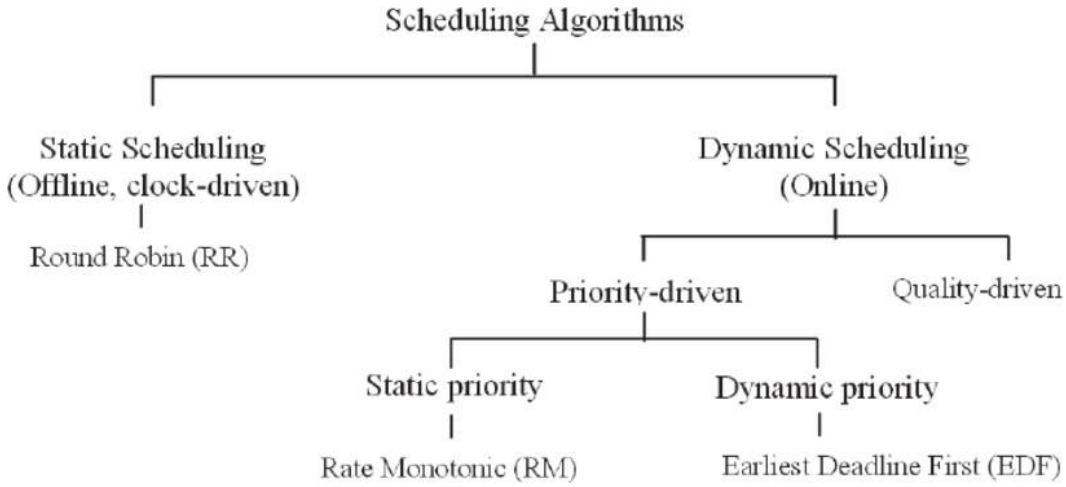


Fig:2.2:Classification of scheduling algorithm

Offline Scheduling Algorithm

Offline scheduling algorithm selects a task to execute with reference to a predetermined schedule, which repeats itself after specific interval of time. For example, if we have three tasks T_a, T_b and T_c then T_a will always execute first, then T_b and after that T_c respectively.

Online Scheduling Algorithm

In Online scheduling a task executes with respect to its priority, which is determined in real time according to specific rule and priorities of tasks may change during execution. The online line scheduling algorithm has two types. They are more flexible because they can change the priority of tasks on run time according to the utilization factor of tasks.

Fixed priority algorithms

In fixed priority if the k^{th} job of a task T1 has higher priority than the k^{th} job of task T2 according to some specified scheduling event, then every job of T1 will always execute first then the job of T2 i.e. on next occurrence priority does not change. More formally, if job J(1,K) of task T1 has higher priority than J(2,K) of task T2 then J(1,K+1) will always has higher priority than of J(2,K+1) . One of best example of fixed priority algorithm is rate monotonic scheduling algorithm.

Dynamic priority algorithms

In dynamic priority algorithm, different jobs of a task may have different priority on next occurrence, it may be higher or it may be lower than

the other tasks. One example of a dynamic priority algorithm is the earliest deadline first algorithm.

Processor utilization factor (U)

For a given task set of n periodic tasks, processor utilization factor U is the fraction of time that is spent for the execution of the task set. If Si is a task from task set then Ci/Ti is the time spent by the processor for the execution of Si . Processor utilization factor is denoted as

$$U=\sum_{i=1}^{n} \frac{C_i}{T_i}$$

Similarly, for the task set of n periodic tasks processor utilization is greater than one then that task set will not be schedulable by any algorithm. Processor utilization factor tells about the processor load on a single processor. U=1 means 100% processor utilization. Following scheduling algorithms will be discussed in details

Rate Monotonic (RM) Scheduling Algorithm

The Rate Monotonic scheduling algorithm is a simple rule that assigns priorities to different tasks according to their time period. That is task with smallest time period will have highest priority and a task with longest time period will have lowest priority for execution. As the time period of a task does not change so not its priority changes over time, therefore Rate monotonic is fixed priority algorithm. The priorities are decided before the start of execution and they does not change overtime.

Rate monotonic scheduling Algorithm works on the principle of preemption. Preemption occurs on a given processor when higher priority task blocked lower priority task from execution. This blocking occurs due to priority level of different tasks in a given task set. rate monotonic is a preemptive algorithm which means if a task with shorter period comes during execution it will gain a higher priority and can block or preemptive currently running tasks. In RM priorities are assigned according to time period. Priority of a task is inversely proportional to its timer period.

Task with lowest time period has highest priority and the task with highest period will have lowest priority. A given task is scheduled under rate monotonic scheduling Algorithm, if its satisfies the following equation:

Example of Rate Monotonic (RM) Scheduling Algorithm

For example, we have a task set that consists of three tasks as follows

Table 2.1 Rate Monotonic (RM) Scheduling Algorithm

Tasks	Release time(ri)	Execution time(Ci)	Deadline (Di)	Time period(Ti)
T1	0	0.5	3	3
T2	0	1	4	4
T3	0	2	6	6

Task set U= 0.5/3 +1/4 +2/6 = 0.167+ 0.25 + 0.333 = 0.75

As processor utilization is less than 1 or 100% so task set is schedulable and it also satisfies the above equation of rate monotonic scheduling algorithm.

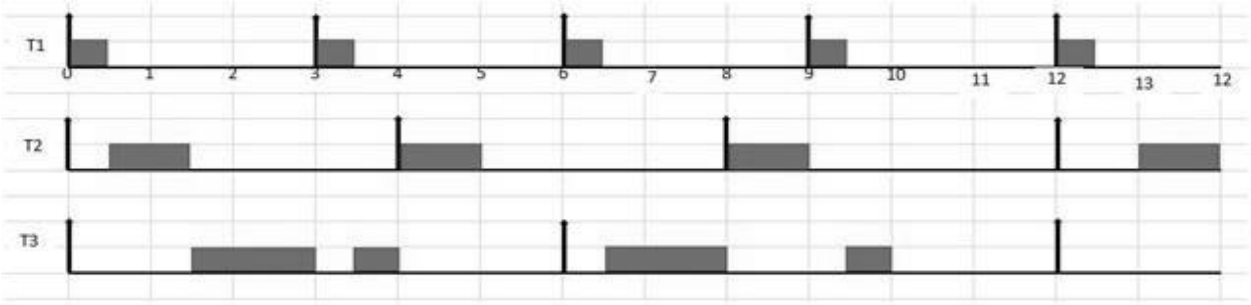


Fig:2.3: RM scheduling of Task set in Table 2.1

A task set given in the above table is RM scheduling in the given figure. The explanation of above is as follows

1. **According to RM scheduling algorithm task with shorter period has higher priority so T1 has high priority, T2 has intermediate priority and T3 has lowest priority. At t=0 all the tasks are released. Now T1 has highest priority so it executes first till t=0.5.**
2. **At t=0.5 task T2 has higher priority than T3 so it executes first for one-time units till t=1.5. After its completion only one task is remained in the system that is T3, so it starts its execution and executes till t=3.**
3. **At t=3 T1 releases, as it has higher priority than T3 so it preempts or blocks T3 and starts it execution till t=3.5. After that the remaining part of T3 executes.**
4. **At t=4 T2 releases and completes it execution as there is no task running in the system at this time.**
5. **At t=6 both T1 and T3 are released at the same time but T1 has higher priority due to shorter period so it preempts T3 and executes till t=6.5, after that T3 starts running and executes till t=8.**

6. **At t=8 T2 with higher priority than T3 releases so it preempts T3 and starts its execution.**
7. **At t=9 T1 is released again and it preempts T3 and executes first and at t=9.5 T3 executes its remaining part. Similarly, the execution goes on.**

Advantages

- **It is easy to implement.**
- **If any static priority assignment algorithm can meet the deadlines then rate monotonic scheduling can also do the same. It is optimal.**
- **It consists of calculated copy of the time periods unlike other time-sharing algorithms as Round robin which neglects the scheduling needs of the processes.**

Disadvantages

- **It is very difficult to support a periodic and sporadic tasks under RMA.**
- **RMA is not optimal when tasks period and deadline differ.**

Least laxity scheduling

Least Laxity First (LLF) is a job level dynamic priority scheduling algorithm. It means that every instant is a scheduling event because laxity of each task changes on every instant of time. A task which has least laxity at an instant, it will have higher priority than others at this instant.

Introduction to Least Laxity first (LLF) scheduling Algorithm

More formally, priority of a task is inversely proportional to its run time laxity. As the laxity of a task is defined as its urgency to execute. Mathematically it is described as

$$L_i = D_i - C_i$$

$$L_i = D_i - (t_i + C_i^R)$$

Here D_i is the deadline of a task, C_i is the worst-case execution time(WCET) and Li
is laxity of a task. It means laxity is the time remaining after completing the WCET before the deadline arrives. For finding the laxity of a task in run time current instant of time also included in the above formula.

Here is the current instant of time and is the remaining WCET of the task. By using the above equation laxity of each task is calculated at every instant of time, then the priority is assigned. One important thing to note is that

laxity of a running task does not changes it remains same whereas the laxity all other tasks is decreased by one after every one-time unit.

Example of Least Laxity first scheduling Algorithm

An example of LLF is given below for a task set.

Table 2.2 Least Laxity first scheduling Algorithm

Task	Release time(ri)	Execution Time(Ci)	Deadline (Di)	Period(Ti)
T1	0	2	6	6
T2	0	2	8	8
T3	0	3	10	10

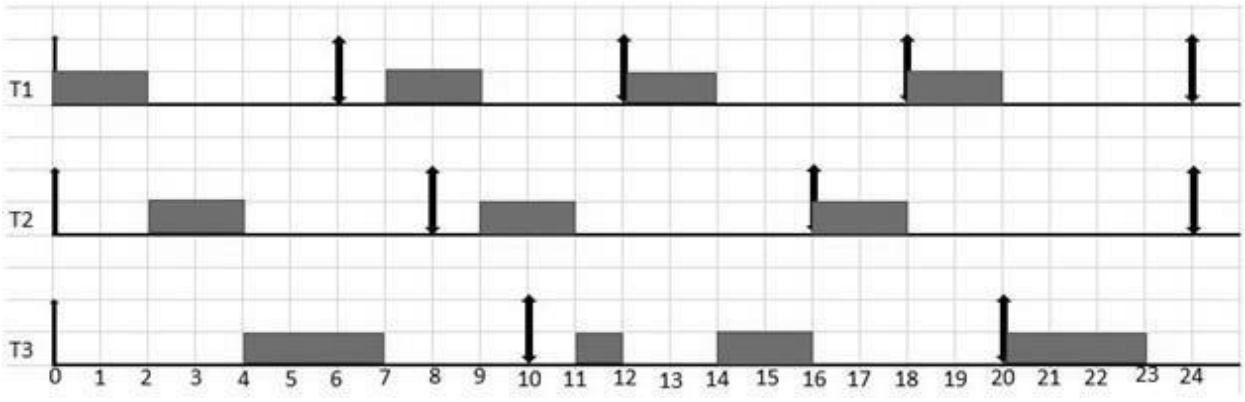


Fig:2.4: LLF scheduling algorithm

1. **At t=0 laxities of each task are calculated by using the equation**

L1 = 6-(0+2) =4
L2 = 8-(0+2) =6

L3= 10-(0+3) =7

As task T1 has least laxity so it will execute with higher priority. Similarly, At t=1 its priority is calculated it is 4 and T2 has 5 and T3 has 6, so again due to least laxity T1 continue to execute.

2. **At t=2 T1is out of the system so Now we compare the laxities of T2 and T3 as following**

$$L2= 8-(2+2) =4$$

$$L3= 10-(2+3) =5$$

Clearly T2 starts execution due to less laxity than T3. At t=3 T2 has laxity 4 and T3 also has laxity 4, so ties are broken randomly so we continue to execute T2. At t=4 no task is remained in the system except T3 so it executes till t=6. At t=6 T1 comes in the system so laxities are calculated again

$$L1 = 6-(0+2) =4$$

$$L3= 10-(6+1) =3$$

So T3 continues its execution.

3. **At t=8 T2 comes in the system where as T1 is running task. So at this instant laxities are calculated**

$$L1 = 12-(8+1) =3$$
$$L2= 16-(8+2) =6$$

So T1 completes its execution. After that T2 starts running and at t=10 due to laxity comparison T2 has higher priority than T3 so it completes it execution.

$$L2= 16-(10+1) =5$$

$$L3= 20-(10+3) =7$$

t=11 only T3 in the system so it starts its execution.

4. **At t=12 T1 comes in the system and due to laxity comparison at t=12 T1 wins the priority and starts its execution by preempting T3. T1 completes it execution and after that at t=14 due to alone task T3 starts running its remaining part. So, in this way this task set executes under LLF algorithm.**

LLF is an optimal algorithm because if a task set will pass utilization test then it is surely schedulable by LLF. Another advantage of LLF is that it some advance knowledge about which task going to miss its deadline. On other hand it also has some disadvantages as well one is its enormous computation demand as each time instant is a scheduling event. It gives poor performance when more than one task has least laxity.

Cyclic executives:

- **Scheduling tables**
- **Frames**
- **Frame size constraints**
- **Generating schedules**
- **Non-independent tasks**
- **Pros and cons**

Cyclic Scheduling

This is an important way to sequence tasks in a real time system. Cyclic scheduling is static – computed offline and stored in a table. Task scheduling is non-preemptive. Non-periodic work can be run during time slots not used by periodic tasks. Implicit low priority for non-periodic work. Usually non-periodic work must be scheduled preemptively. Scheduling table executes completely in one hyper period H. Then repeats H is least common multiple of all task periods N quanta per hyper period. Multiple tables can support multiple system modes E.g., an aircraft might support takeoff, cruising, landing, and taxiing modes. Mode switches permitted only at hyper period boundaries. Otherwise, hard to meet deadlines.

$$T(t_k) = \begin{cases} T_i & \text{if } T_i \text{ is to be scheduled at time } t_k \\ I & \text{if no periodic task is scheduled at time } t_k \end{cases}$$

Frames:

Divide hyper periods into frames .Timing is enforced only at frame boundaries.

Consider a system with four task

- $T_1 = (4,1)$
- $T_2 = (5, 1.8)$
- $T_3 = (20, 1)$
- $T_4 = (20, 2)$

Possible schedule:

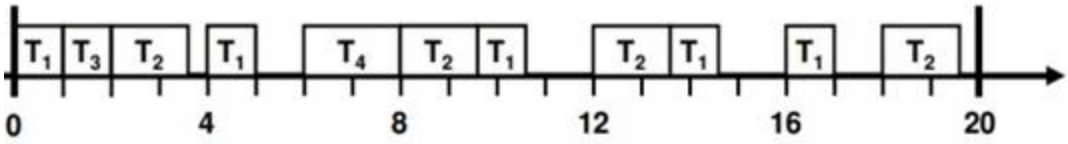


Table starts out with:

- $(0, T_1), (1, T_3), (2, T_2), (3.8, I), (4, T_1), \ldots$

task is executed as a function call and must fit within a single frame. Multiple tasks may be executed in a frame size is f Number of frames per hyper period is F = H/f.

Frame Size Constraints:

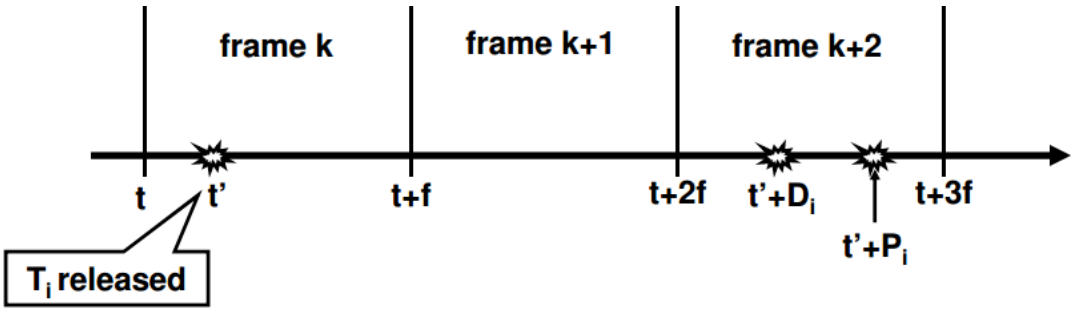
1. Tasks must fit into frames so, f ≥ Ci for all tasks Justification: Non-preemptive tasks should finish executing within a single frame

2. f must evenly divide H Equivalently, f must evenly divide P for some task i

Justification: Keep table size small

3. There should be a complete frame between the release and deadline of every task

Justification: Want to detect missed deadlines by the time the deadline arrives



- **Therefore: $2f - \gcd(P_i, f) \leq D_i$ for each task i**

Cyclic executive is one of the major software architectures for embedded systems. Historically, cyclic executives dominate safety-critical systems Simplicity and predictability wins. However, there are significant drawbacks. Finding a schedule might require significant offline computation.

Drawbacks:

- **Difficult to incorporate sporadic processes.**
- **Incorporate processes with long periods**
- **Problematic for tasks with dependencies – Think about an OS without synchronization**
- **Time consuming to construct the cyclic executive – Or adding a new task to the task set**
- **"Manual" scheduler construction**
- **Cannot deal with any runtime changes**
- **Denies the advantages of concurrent programming**

Summary:

- **Off-line scheduling**
- **Doesn't use the process abstraction of the OS**
- **Manually created table of procedures to be called – Waits for a periodic interrupt for synchronization**
- **Minor cycle – Loops the execution of the procedures in the table**

TEXT / REFERENCE BOOKS

1. Jane W. S Liu, "Real Time Systems" Pearson Higher Education, 3rd Edition, 2000.

2. Raj Kamal, "Embedded Systems- Architecture, Programming and Design" Tata McGraw Hill, 2nd Edition, 2014.

3. Jean J. Labrosse, "Micro C/OS-I : The real time kernel" CMP Books, 2nd Edition,2015.

5. Richard Barry, "Mastering the Free RTOS: Real Time Kernel", Real Time Engineers Ltd, 1st Edition, 2016.

6. David E. Simon, " An Embedded Software Primer", Pearson Education Asia Publication

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT - III

RTOS PROGRAMMING – SECA5302

III RESOURCE SHARING FOR REAL TIME TASKS

Resource sharing among tasks- Priority inversion Problem - Priority inheritance and Priority ceiling Protocols – Features of commercial and open source real time operating systems: Vxworks, QNX, Micrium OS, RT Linux and Free RTOS

Sharing of critical resources among tasks requires a different set of rules, compared to the rules used for sharing resources such as a CPU among tasks. We have in the last Chapter discussed how resources such as CPU can be shared among tasks. Priority inversion is a operating system scenario in which a higher priority process is preempted by a lower priority process. This implies the inversion of the priorities of the two processes.

Problems due to Priority Inversion

Some of the problems that occur due to priority inversion are given as follows −

- **A system malfunction may occur if a high priority process is not provided the required resources.**
- **Priority inversion may also lead to implementation of corrective measures. These may include the resetting of the entire system.**
- **The performance of the system can be reduces due to priority inversion. This may happen because it is imperative for higher priority tasks to execute promptly.**
- **System responsiveness decreases as high priority tasks may have strict time constraints or real time response guarantees.**
- **Sometimes there is no harm caused by priority inversion as the late execution of the high priority process is not noticed by the system.**

Solutions of Priority Inversion

Some of the solutions to handle priority inversion are given as follows −

- **Priority Ceiling**

  All of the resources are assigned a priority that is equal to the highest priority of any task that may attempt to claim them. This helps in avoiding priority inversion

- **Disabling Interrupts**

  There are only two priorities in this case i.e. interrupts disabled and preemptible. So priority inversion is impossible as there is no third option.

- **Priority Inheritance**

  This solution temporarily elevates the priority of the low priority task that is executing to the highest priority task that needs the resource. This means that medium priority tasks cannot intervene and lead to priority inversion.

- **No blocking**

  Priority inversion can be avoided by avoiding blocking as the low priority task

blocks the high priority task.

- **Random boosting**

  The priority of the ready tasks can be randomly boosted until they exit the critical section.

Difference between Priority Inversion and Priority Inheritance

Both of these concepts come under Priority scheduling in Operating System. In one line, Priority Inversion is a problem while *Priority Inheritance* is a solution. Literally, Priority Inversion means that priority of tasks get inverted and *Priority Inheritance* means that priority of tasks get inherited. Both of these phenomena happen in priority scheduling. Basically, in *Priority Inversion*, higher priority task (H) ends up waiting for middle priority task (M) when H is sharing critical section with lower priority task (L) and L is already in critical section. Effectively, H waiting for M results in inverted priority i.e. Priority Inversion. One of the solution for this problem is Priority Inheritance.

In Priority Inheritance, when L is in critical section, L inherits priority of H at the time when H starts pending for critical section. By doing so, M doesn't interrupt L and H doesn't wait for M to finish. Please note that inheriting of priority is done temporarily i.e. L goes back to its old priority when L comes out of critical section.

Priority Inheritance Protocol (PIP)

Priority Inheritance Protocol (PIP) is a technique which is used for sharing critical resources among different tasks. This allows the sharing of critical resources among different without the occurrence of unbounded priority inversions.

Basic Concept of PIP :

The basic concept of PIP is that when a task goes through priority inversion, the priority of the lower priority task which has the critical resource is increased by the priority inheritance mechanism. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion.

Working of PIP :

When several tasks are waiting for the same critical resource, the task which is currently holding this critical resource is given the highest priority among all the tasks which are waiting for the same critical resource. Now after the lower priority task having the critical resource is given the highest priority then the intermediate priority tasks cannot preempt this task. This helps in avoiding the unbounded priority inversion. When the task which is given the highest priority among all tasks, finishes the job and releases the critical resource then it gets back to its original priority value (which may be less or equal). If a task is holding multiple critical resources then after releasing one critical resource it cannot go back to it original priority value. In this case it inherits the highest priority among all tasks waiting for the same critical resource.

Advantages of PIP :

Priority Inheritance protocol has the following advantages:

- **It allows the different priority tasks to share the critical resources.**
- **The most prominent advantage with Priority Inheritance Protocol is that it avoids the unbounded priority inversion.**

Disadvantages of PIP :

Priority Inheritance Protocol has two major problems which may occur:

Deadlock :

There is possibility of deadlock in the priority inheritance protocol. For example, there are two tasks T_1 and T_2. Suppose T_1 has the higher priority than T_2. T_2 starts running first and holds the critical resource CR_2. After that, T_1 arrives and preempts T_2. T_1 holds critical resource CR_1 and also tries to hold CR_2 which is held by T_2. Now T_1 blocks and T_2 inherits the priority of T_1 according to PIP. T_2 starts execution and now T_2 tries to hold CR_1 which is held by T_1. Thus, both T_1 and T_2 are deadlocked.

Chain Blocking :

When a task goes through priority inversion each time it needs a resource then this process is called chain blocking. For example, there are two tasks T_1 and T_2. Suppose T_1 has the higher priority than T_2. T_2 holds the critical resource CR_1 and CR_2. T_1 arrives and requests for CR_1. T_2 undergoes the priority inversion according to PIP. Now, T_1 request CR_2, again T_2 goes for priority inversion according to PIP. Hence, multiple priority inversion to hold the critical resource leads to chain blocking.

Priority Ceiling Protocol

In real-time computing, the priority ceiling protocol is a synchronization protocol for shared resources to avoid unbounded priority inversion and mutual deadlock due to wrong nesting of critical sections. In this protocol each resource is assigned a priority ceiling, which is a priority equal to the highest priority of any task which may lock the resource. The protocol works by temporarily raising the priorities of tasks in certain situations, thus it requires a scheduler that supports dynamic priority scheduling. It is a job task synchronization protocol in a real-time system that is better than Priority inheritance protocol in many ways. Real-Time Systems are multitasking systems that involve the use of semaphore variables, signals, and events for job synchronization. In Priority ceiling protocol an assumption is made that all the jobs in the system have a fixed priority. It does not fall into a deadlock state.

The chained blocking problem of the Priority Inheritance Protocol is resolved in the Priority Ceiling Protocol.

The basic properties of Priority Ceiling Protocols are:

1. **Each of the resources in the system is assigned a priority ceiling.**
2. **The assigned priority ceiling is determined by the highest priority among all the jobs which may acquire the resource.**
3. **It makes use of more than one resource or semaphore variable, thus eliminating chain blocking.**
4. **A job is assigned a lock on a resource if no other job has acquired lock on that resource.**
5. **A job J, can acquire a lock only if the job's priority is strictly greater than the priority ceilings of all the locks held by other jobs.**
6. **If a high priority job has been blocked by a resource, then the job holding that resource gets the priority of the high priority task.**
7. **Once the resource is released, the priority is reset back to the original.**
8. **In the worst case, the highest priority job J_1 can be blocked by T lower priority tasks in the system when J_1 has to access T semaphores to finish its execution.**

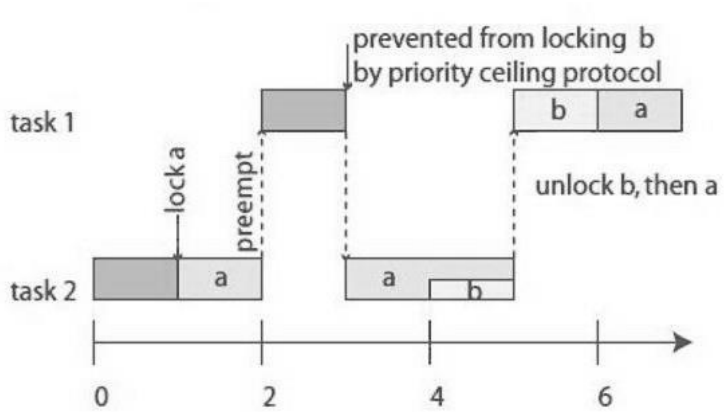


Fig:3.1: Priority Ceiling Protocol

Priority Scheduling Protocol can be used to tackle the problem of the priority inversion problem unlike that of Priority Inheritance Protocol. It makes use of semaphores to share the resources with the jobs in a real-time system.

Features of firmware and commercial real time operating systems:

Vx Works features:

- **High –performance**
- **Unix performance**
- **Unix -like, multitasking**
- **Environment scalable and hierarchical RTOS**
- **Hierarchical RTOS**
- **Host and target based development approach Supports**
- **Device Software Optimization ─ a new methodology that enables development and running of device software faster, better and more reliably Vx works RTOS Kernel**
- **VxWorks 6.x processor abstraction layer**

- **The layer enables application design for new versions later by just changing the layer hardware**
- **interface**
- **Supports advanced processor architectures ─ ARM, Cold Fire, MIPS, Intel, SuperH.**
- **Hard real time applications**
- **Supports kernel mode execution of Supports kernel mode execution of tasks**
- **Supports open source Linux and TIPC protocol**
- **Provides for the preemption points at kernel**
- **Provides preemptive as well as round robin scheduling**
- **Support POSIX standard asynchronous IOs**
- **Support UNIX standard buffered I/Os**
- **PTTS 1.1 (Since Dec. 2007)**
- **IPCs in TIPC for network and clustered system environment**
- **POSIX 1003.1b standard IPCs and interfaces additional availability**
- **Separate context for tasks and ISRs**

Micrium OS:

µC/OS is a full-featured embedded operating system. Features support for TCP/IP, USB, CAN bus, and Modbus. Includes a robust file system, and graphical user interface

Reliable: Micrium software includes comprehensive documentation, full source code, powerful debugging features, and support for a huge range of CPU architectures. Micrium software offers unprecedented ease-of-use, a small memory footprint, remarkable energy efficiency, and all with a full suite of protocol stacks.Real-Time Kernels

Portable. Offering unprecedented ease-of-use, µC/OS kernels are delivered with complete source code and in-depth documentation. The µC/OS kernels run on huge number of processor architectures

Scalable. The µC/OS kernels allow for unlimited tasks and kernel objects. The kernels' memory footprint can be scaled down to contain only the features required for your application, typically 6–24 KBytes of code space and 1 KByte of data space.

Efficient. Micrium's kernels also include valuable runtime statistics, making the internals of your application observable. Identify performance bottlenecks, and optimize power usage, early in your development cycle.
The features of the µC/OS kernels include:

- **Preemptive multitasking real-time kernel with optional round robin scheduling**
- **Delivered with complete, clean, consistent source code with in-depth documentation.**

- **Highly scalable: Unlimited number of tasks, priorities and kernel objects**
- **Resource-efficient: 6K to 24K bytes code space, 1K+ bytes data space)**
- **Very low interrupt disable time**
- **Extensive performance measurement metrics (configurable)**
- **Certifiable for safety-critical applications**
- **Micrium provides two extensions to the µC/OS-II kernel that provide memory protection and greater stability and safety for the applications.**

µC/OS-MPU is an extension for Micrium's µC/OS-II kernel that provides memory protection. The µC/OS-MPU extension prevents applications from accessing forbidden locations, thereby protecting against damage to safety-critical applications, such as medical devices and avionics systems. µC/OS-MPU builds a system with MPU processes, with each containing one or more tasks or threads. Each process has an individual read, write, and execution right. Exchanging data between threads is accomplished in the same manner using µC/OS-II tasks, however handling across different processes is achieved by the core operating system.

µC/OS-MPU includes the following features:

- **Prevents access of forbidden locations**
- **Appropriate for safety-critical applications**
- **Easy integration of protocol stacks, graphical modules, FS libraries**
- **Simplified debugging and error diagnosis**
- **Available for any microcontroller equipped with a hardware Memory Protection Unit (MPU) or Memory Management Unit (MMU).**
- **Third-party certification support available**

Debugging and error diagnosis is simplified as an error management system provides information on the different processes. The hardware protection mechanism cannot be bypassed by software. Existing µC/OS-applications can be adapted with minimal effort. µC/OS-MPU is available for any microcontroller containing a Memory Protection Unit (MPU) or Memory Management Unit (MMU). Third-party certification support is also available. µC/TimeSpaceOS is an extension for Micrium's µC/OS-II kernel that manages both memory and time allocated to diverse types of applications. With µC/TimeSpaceOS you can certify complex systems for safety-critical applications cost-effectively.

Its features and benefits include:

- **Memory protection so that multiple applications cannot influence, disturb or interact with each other**
- **Deterministic and run-time guaranteed**
- **Configurable so that virtual Dynamic Random Access Memory (DRAM) is optionally available**
- **A small footprint for use in a wide-range of applications**
- **Compatibility: can be used within the protected segment in existing µC/OS-II applications**
- **Certification according to DO178B and IEC61508**
- **Available for a large number of microcontrollers; contact us for details**

µC/TimeSpaceOS makes it possible for several independent applications (with or without real-time kernels) within one environment to be executed on one target hardware platform. It guarantees that the applications will not influence or interfere with each other. Each application is developed in a protected memory area (partition). The application is independent with respect to other partitions. This makes it easier for multiple developers to develop complex control devices. Each partition can be considered its own virtual CPU.

Features of RT Linux:

- **Multi-tasking**
- **Priority-based scheduling**
- **Application tasks should be programmed to suit**
- **Ability to quickly respond to external interrupts**
- **Basic mechanisms for process communication and synchronization**
- **Small kernel and fast context switch**

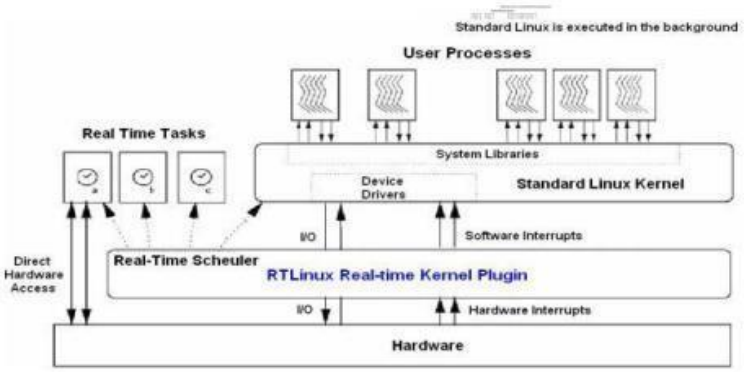


Fig:3.2: Features of RT Linux

Priority based kernel for embedded applications e.g. OSE, Vx Works, QNX, VRTX32, pSOS . Many of them are commercial kernels . Applications should be designed and programmed to suite priority-based scheduling e.g deadlines as priority etc . Real Time Extensions of existing time-sharing OS. e.g. Real time Linux, Real time NT by e.g locking RT tasks in main memory, assigning highest priorities etc .Research RT Kernels e.g. SHARK, TinyOS . Run-time systems for RT programming languages e.g. Ada, Erlang, Real-Time Java

Free RTOS and C Executive:

Free RTOS is a popular real-time operating system kernel for embedded devices, which has been ported to 35 microcontrollers. It is distributed under the GPL with an additional restriction and optional exception. The restriction forbids benchmarking while the exception permits users' proprietary code to remain closed source while maintaining the kernel itself as open source, thereby facilitating the use of Free RTOS in proprietary applications.

Free RTOS is designed to be small and simple. The kernel itself consists of only three or four C files. To make the code readable, easy to port, and maintainable, it is written mostly in C, but there are a few assembly functions included where needed (mostly in architecture-specific scheduler routines). Thread priorities are supported. In addition there are four schemes of memory allocation provided:

Allocate only;

- **Allocate and free with a very simple, fast, algorithm;**
- **A more complex but fast allocate and free algorithm with memory coalescence;**
- **C library allocate and free with some mutual exclusion protection**

Key features:

- **Very small memory footprint, low overhead, and very fast execution.**
- **Tick-less option for low power applications.**
- **Equally good for hobbyists who are new to OSes, and professional developers working on commercial products.**
- **Scheduler can be configured for either preemptive or cooperative operation.**
- **Co routine support (Co routine in Free RTOS is a very simple and lightweight task that has very limited use of stack)**
- **Trace support through generic trace macros. Tools such as Trace alyzer (a.k.a. Free RTOS+Trace, provided by the Free RTOS partner Percepio) can thereby record and visualize the runtime behavior of Free RTOS-based systems. This includes task scheduling and kernel calls for semaphore and queue operations. Trace analyzer is a commercial tool, but also available in a feature-limited free version.**

Features of a RTOS:

- **Allows multi-tasking**
- **Scheduling of the tasks with priorities**

- **Synchronization of the resource access**
- **Inter-task communication**
- **Time predictable**
- **Interrupt handling**

Predictability of timing

- **The timing behavior of the OS must be predictable**
- **For all services of the OS, there is an upper bound on the execution time**
- **Scheduling policy must be deterministic**
- **The period during which interrupts are disabled must be short (to avoid unpredictable delays in the processing of critical events)**

QNX RTOS v6.1

The QNX RTOS v6.1 has a client-server based architecture. QNX adopts the approach of implementing an OS with a 10 Kbytes micro-kernel surrounded by a team of optional processes that provide higher-level OS services .Every process including the device driver has its own virtual memory space. The system can be distributed over several nodes, and is network transparent. The system performance is fast and predictable and is robust. It supports Intel x86family of processors, MIPS, PowerPC, and Strong ARM .

QNX has successfully been used in tiny ROM-based embedded systems and in several hundred node distributed systems.

VxWorks (Wind River Systems)

VxWorks is the premier development and execution environment for complex real-time and embedded applications on a wide variety of target processors. Three highly integrated components are included with Vxworks: a high performance scalable real-time operating system which executes on a target processor; a set of powerful cross-development tools; and a full range of communications software options such as Ethernet or serial line for the target connection to the host. The heart of the OS is the Wind microkernel which supports multitasking, scheduling, inter task management and memory management. All other functionalities are through processes. There is no privilege protection between system and application and also the support for communication between processes on different processors is poor.

TEXT / REFERENCE BOOKS

1. Jane W. S Liu, "Real Time Systems" Pearson Higher Education, 3rd Edition, 2000.
2. Raj Kamal, "Embedded Systems- Architecture, Programming and Design" Tata McGraw Hill, 2nd Edition, 2014.
3. Jean J. Labrosse, "Micro C/OS-I : The real time kernel" CMP Books, 2nd Edition,2015.
5. Richard Barry, "Mastering the Free RTOS: Real Time Kernel", Real Time Engineers Ltd, 1st Edition, 2016.
6. David E. Simon, " An Embedded Software Primer", Pearson Education Asia Publication

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT -IV

RTOS PROGRAMMING – SECA5302

IV APPLICATION PROGRAMMING USING RTOS

Task synchronization using semaphores, Inter task communication: message queues and pipes, Remote procedure call- Timers and Interrupts-Memory management and I/O management

Task Management:

Task is the basic notion in RTOS.

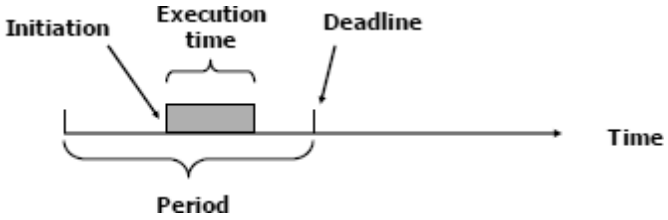


Fig:4.1:Typical RTOS Task Model

Periodic tasks: arriving at fixed frequency, can be characterized by 3 parameters (C,D,T) where, C = computing time, D = deadline and T = period. Also called Time-driven tasks, their activations are generated by timers.

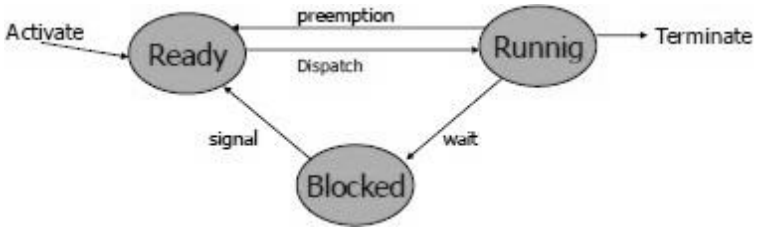


Fig:4.2:Task states

Non-Periodic or aperiodic tasks = all tasks that are not periodic, also known as Event driven, their activations may be generated by external interrupts. Sporadic tasks= aperiodic tasks with minimum inter arrival time T_{min}.

Managing tasks:

- **Execution of quasi-parallel tasks on a processor using processes or threads by maintaining process states, process queuing, allowing for preemptive tasks and quick interrupt handling.**
- **CPU scheduling**
- **Process synchronization**
- **Inter-process communication**
- **Support of a real-time clock as an internal time reference**

- **Task creation: create a new TCB (task control block)**
- **Task termination: remove the TCB**
- **Change Priority: modify the TCB**
- **State-inquiry: read the TCB**

Task management is depicted in the below diagram

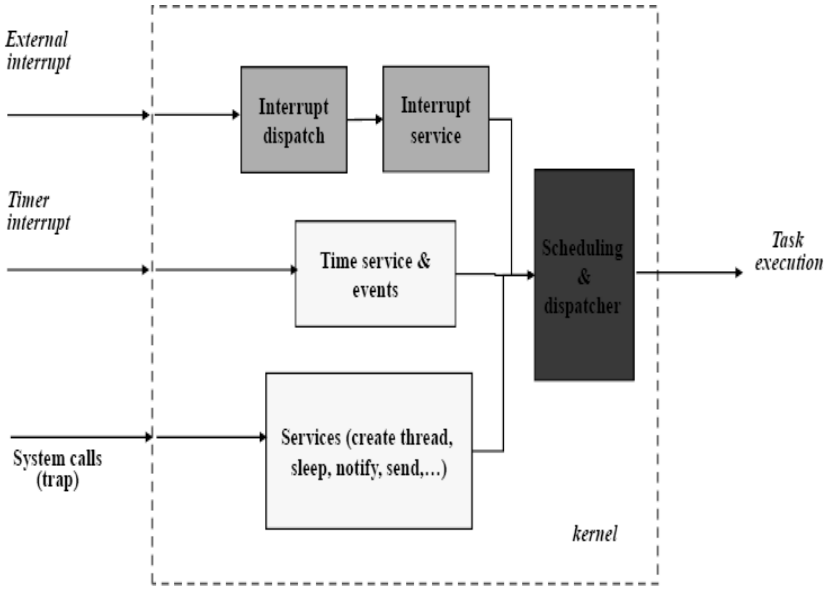


Fig:4.3: Block Diagram of Task management

Task synchronization:

In classical operating systems, synchronization and mutual exclusion is performed via semaphores and monitors. In real-time OS, special semaphores and a deep Integration into scheduling is necessary (priority inheritance protocols).

Further responsibilities: Initializations of internal data structures (tables, queues, task description blocks, semaphores)

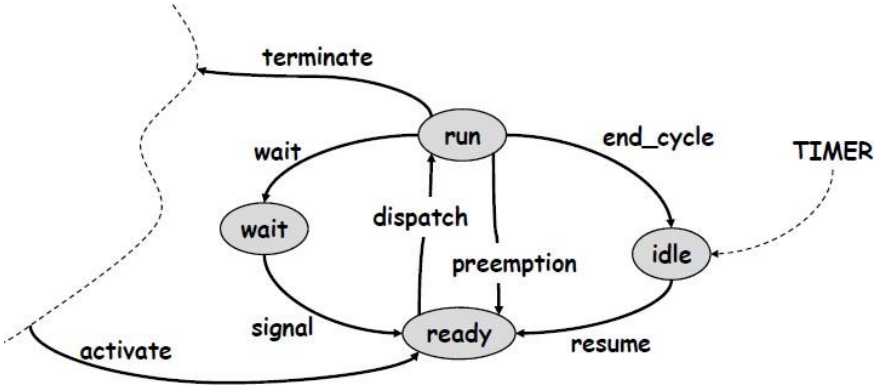


Fig:4.4: Minimal Set of Task States

Run:

A task enters this state as it starts executing on the processor

Ready:

State of those tasks that are ready to execute but cannot be executed, because the processor is assigned to another task.

Wait:

A task enters this state when it executes a synchronization primitive to wait for an event, e.g. a wait primitive on a semaphore. In this case, the task is inserted in a queue associated with the semaphore. The task at the head is resumed when the semaphore is unlocked by a signal primitive.

Idle:

A periodic job enters this state when it completes its execution and has to wait for the beginning of the next period.

Challenges for an RTOS:

- **Creating an RT task, it has to get the memory without delay: this is difficult because memory has to be allocated and a lot of data structures, code segment must be copied/initialized**
- **The memory blocks for RT tasks must be locked in main memory to avoid access latencies due to swapping**
- **Changing run-time priorities dangerous: it may change the run-time behavior and predictability of the whole system**

Inter task communication and Synchronization:

Inter process Communication (IPC) enables processes to communicate with each other to share information

- **Pipes (half duplex)**
- **FIFOs(named pipes)**
- **Stream pipes (full duplex)**
- **Named stream pipes**
- **Message queues**
- **Semaphores**
- **Shared Memory**
- **Sockets**
- **Streams**
- **Pipes**

Oldest (and perhaps simplest) form of UNIX IPC

- **Half duplex**
- **Data flows in only one direction**
- **Only usable between processes with a common ancestor**
- **Usually parent-child**
- **Also child-child**

In computing, a named pipe (also known as a FIFO) is one of the methods for intern-process communication.

- **It is an extension to the traditional pipe concept on Unix. A traditional pipe is "unnamed" and lasts only as long as the process.**
- **A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.**
- **Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.**
- **A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.**

Understanding Pipes

- **Within a process**
- **Writes to files can be read on files**
- **Not very useful**
- **Between processes**
- **After a fork()**
- **Writes to files by one process can be read on files by the other**

Using Pipes:

- **Usually, the unused end of the pipe is closed by the process**
- **If process A is writing and process B is reading, then process A would close files[0] and process B would close files[1]**
- **Reading from a pipe whose write end has been closed returns 0 (end of file)**
- **Writing to a pipe whose read end has been closed generates SIGPIPE**
- **PIPE_BUF specifies kernel pipe buffer size**

Creating a Pipe

The primitive for creating a pipe is the pipe function. This creates both the reading and writing ends of the pipe. It is not very useful for a single process to use a pipe to talk to itself. In typical use, a process creates a pipe just before it forks one or more child processes. The pipe is then used for communication either between the parent or child processes, or between two sibling processes.

The pipe function is declared in the header file unistd.h. Here is an example of a simple program that creates a pipe. The parent process writes data to the pipe, which is read by the child process.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

/* Read characters from the pipe and echo them to stdout. */

void
read_from_pipe (int file)
{
  FILE *stream;
  int c;
  stream = fdopen (file, "r");
  while ((c = fgetc (stream)) != EOF)
    putchar (c);
  fclose (stream);
}

/* Write some random text to the pipe. */

void
write_to_pipe (int file)
{
  FILE *stream;
  stream = fdopen (file, "w");
  fprintf (stream, "hello, world!\n");
  fprintf (stream, "goodbye, world!\n");
  fclose (stream);
}

int
main (void)
{
  pid_t pid;
  int mypipe[2];

  /* Create the pipe. */
  if (pipe (mypipe))
    {
      fprintf (stderr, "Pipe failed.\n");
      return EXIT_FAILURE;
    }
```

```
 /* Create the child process. */
 pid = fork ();
 if (pid == (pid_t) 0)
   {

/* This is the child process.
       Close other end first. */
     close (mypipe[1]);
     read_from_pipe (mypipe[0]);
     return EXIT_SUCCESS;
   }
 else if (pid < (pid_t) 0)
   {
     /* The fork failed. */
     fprintf (stderr, "Fork failed.\n");
     return EXIT_FAILURE;
   }
 else
   {
     /* This is the parent process.
        Close other end first. */
     close (mypipe[0]);
     write_to_pipe (mypipe[1]);
     return EXIT_SUCCESS;
   }
}
```

Using Pipes for synchronization and communication:

- **Once you have a pipe or pair of pipes set up, you can use it/them to Signal events (one pipe)**
- **Wait for a message**
- **Synchronize (one or two pipes)**
- **Wait for a message or set of messages**
- **You send me a message when you are ready, then I'll send you a message when I am ready**
- **Communicate (one or two pipes)**
- **Send messages back and forth**
- **popen() and pclose():**

FIFO

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the file system by calling mkfifo.

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

First: Co processes–Nothing more than a process whose input and output are both redirected from another process

FIFOs–named pipes

With regular pipes, only processes with a common ancestor can communicate With FIFOs, any two processes can communicate

Creating and opening a FIFO is just like creating and opening a file

FIFO details:

#include <sys/types.h>

#include <sys/stat.h>

intmkfifo(constchar *pathname, mode_tmode); The modeargument is just like in open()

Can be opened just like a file

When opened, O_NONBLOCK bit is important

Not specified: open() for reading blocks until the FIFO is opened by a writer

Specified: open() returns immediately, but returns an error if opened for writing and no reader exists

Four Queue States:

1. **Both the sending and receiving queue are running.**
2. **Only the sender is executing.**
3. **Sender not executing, but receiver is running.**
4. **Neither the sender nor the receiver are running.**

Parts of Message Queue:

The queue that is local to the sending machine is called the source queue. The local queue of the receiver is called the destination queue. Queues are managed by queue managers. Mapping of queues to network locations.

Message sending methods:

1. **Message Queues**
2. **Mail box:**

Mail box:

Mailboxes are similar to queues.

- **Capacity of mailboxes is usually fixed after initialization**
- **Some RTOS allow only a single message in the mailbox. (mailbox is either full or empty)**
- **Some RTOS allow prioritization of messages**

Mailbox functions at OS

Some OS provide the mailbox and queue both IPC functions. When the IPC functions for mailbox are not provided by an OS, then the OS employs queue for the same purpose. A mailbox of a task can receive from other tasks and has a distinct ID. Mailbox (for message) is an IPC through a message at an OS that can be received only one single destined task for the message from the tasks. Two or more tasks cannot take message from same Mailbox. A task on an OS function call puts (means post and also send) into the mailbox only a pointer to a mailbox message . Mailbox message may also include a header to identify the message-type specification.

OS provides for inserting and deleting message into the mailbox message pointer. Deleting means message-pointer pointing to Null. Each mailbox for a message need initialization (creation) before using the functions in the scheduler for the message queue and message pointer pointing to Null. There may be a provision for multiple mailboxes for the multiple types or destinations of messages. Each mailbox has an ID. Each mailbox usually has one message pointer only, which can point to message.

Mailbox Types

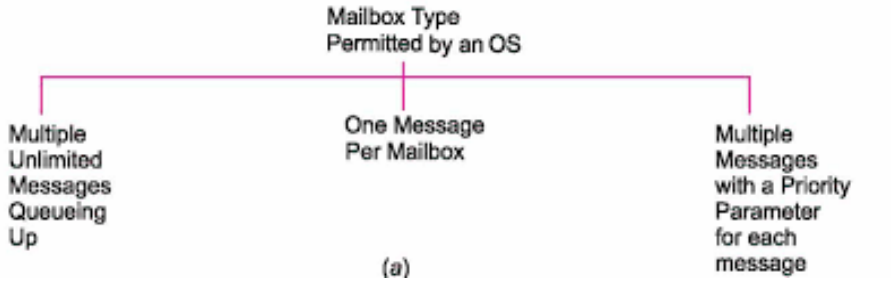


Fig:4.5:Classification of mailbox

Mailbox IPC features

When an OS call is to post into the mailbox, the message bytes are as per the pointed number of bytes by the mailbox message pointer.

Mailbox Related Functions at the OS

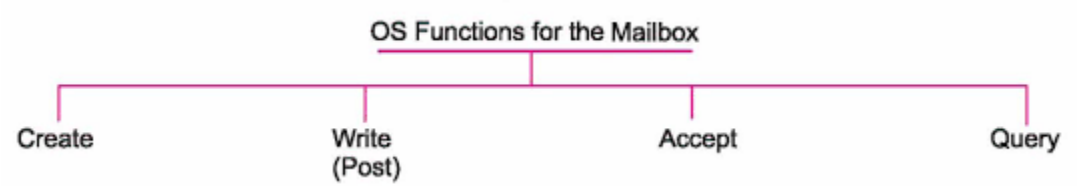


Fig:4.6:Features of mailbox

Mailbox IPC functions

1. **OSMBoxCreate creates a box and initializes the mailbox contents with a NULL pointer at *msg . 2. OSMBoxPost sends at *msg, which now does not point to Null.**
2. **An ISR can also post into mailbox for a task**
3. **OSMBoxWait (Pend) waits for *msg not Null, which is read when not Null and again *msg points to Null. • The time out and error handling function can be provided with Pend function argument. • ISR not permitted to wait for message into mailbox. Only the task can wait**
4. **OSMBoxAccept reads the message at *msg after checking the presence yes or no [No wait.] Deletes (reads) the mailbox message when read and *msg again points to Null**

   • An ISR can also accept mailbox message for a task
5. **OSMBoxQuery queries the mailbox *msg.**
6. **OSMBoxDelete**

Event Management and Memory Management

Tasks are event driven or time driven. In RTOS environment there is less significance for event driven mechanism. Just as processes share the CPU, they also share physical memory. The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

- **Logical address – generated by the CPU; also referred to as virtual address**
- **Physical address – address seen by the memory unit**

Logical and physical addresses are the same in compile-time and load time address binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme. Re locatable Means that the program image can reside anywhere in physical memory. Binding Programs need real memory in which to reside. When is the location of that real memory determined. This is called mapping logical to physical addresses. This binding can be done at compile/link time. Converts symbolic to re locatable. Data used within compiled source is offset within object module.

Compiler: If it's known where the program will reside, then absolute code is generated. Otherwise compiler produces re locatable code.

Load: Binds re locatable to physical. Can find best physical location.

Execution: The code can be moved around during execution. Means flexible virtual Mapping

Example of Memory Usage:

Calculation of an effective address
Fetch from instruction
Use index offset
Example: (Here index is a pointer to an address)
loop:
load register, index
add 42, register store
register, index inc
index
skip_equal index, final_address
branch loop

This binding can be done at compile/link time. Converts symbolic to re locatable. Data used within compiled source is offset within object module. Can be done at load time. Binds reloadable to physical. Can be done at run time. Implies that the code can be moved around during execution.

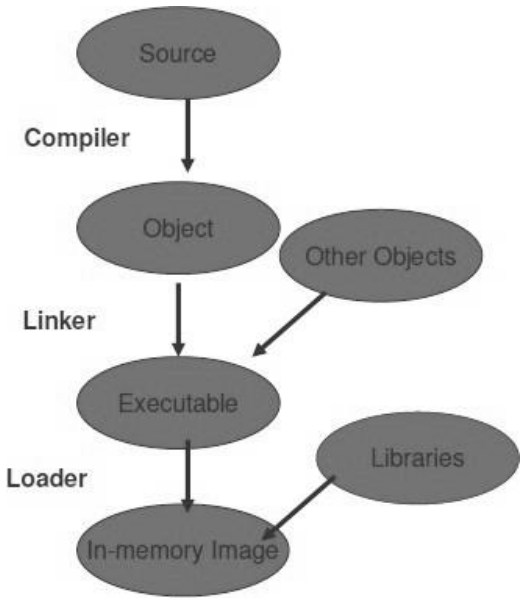


Fig:4.7:Design of software development tools

Message queues

A message queue is an object used for inter task communication through which task send or receive messages placed in a shared memory. The queue may follow
1) First In First Out (FIFO), 2) Last in First Out(LIFO) or 3) Priority (PRI) sequence. Usually, a message queue comprises of an associated queue control block (QCB), name, unique ID, memory buffers, queue length, maximum message length and one or more task waiting lists. A message queue with a length of 1 is commonly known as a mailbox.

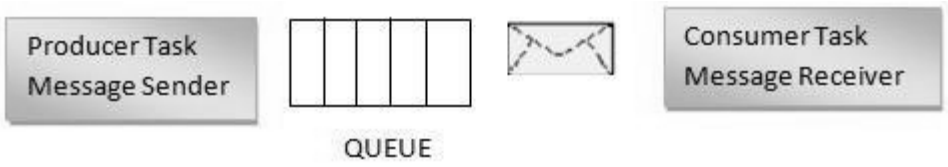


Fig:4.8: Message queues

Pipes

A pipe is an object that provide simple communication channel used for unstructured data exchange among tasks. A pipe does not store multiple messages but stream of bytes. Also, data flow from a pipe cannot be prioritized.

Remote procedure call (RPC)

Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server based applications. It is based on extending the conventional local procedure calling so that the called procedure need not exist in the same address space as the calling procedure. A remote **procedure call is an inter process communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call. A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished**.

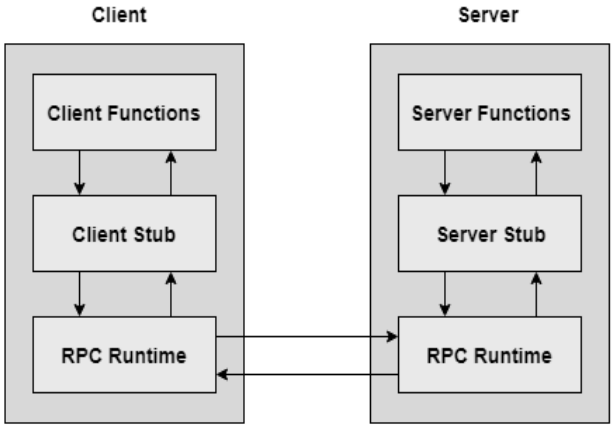


Fig:4.9: Remote procedure call

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

1. A client invokes a client stub procedure, passing parameters in the usual way. The client stub resides within the client's own address space.

2. The client stub marshalls (pack) the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.
3. The client stub passes the message to the transport layer, which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a server stub, which demar shalls(unpack) the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

RPC Issues

1. RPC Runtime: RPC run-time system is a library of routines and a set of services that handle the network communications that underlie the RPC mechanism. In the course of an RPC call, client-side and server-side run-time systems' code handle binding, establish communications over an appropriate protocol, pass call data between the client and server, and handle communications errors.
2. Stub: The function of the stub is to provide transparency to the programmer-written application code. On the client side, the stub handles the interface between the client's local procedure call and the run-time system, marshaling and unmarshaling data, invoking the RPC run-time protocol, and if requested, carrying out some of the binding steps. On the server side, the stub provides a similar interface between the run-time system and the local manager procedures that are executed by the server.
3. Binding: The most flexible solution is to use dynamic binding and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.

Advantages of Remote Procedure Call

Some of the advantages of RPC are as follows

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

Disadvantages of Remote Procedure Call

Some of the disadvantages of RPC are as follows

- **The remote procedure call is a concept that can be implemented in different ways. It is not a standard.**
- **There is no flexibility in RPC for hardware architecture. It is only interaction based.**
- **There is an increase in costs because of remote procedure cal**

Memory Management

Memory management is one of the most important subsystems of any operating system for computer control systems, and is even more critical in a RTOS than in standard operating systems. Firstly, the speed of memory allocation is important in a RTOS. A standard memory allocation scheme scans a linked list of indeterminate length to find a free memory block; however, memory allocation has to occur in a fixed time in a RTOS. Secondly, memory can become fragmented as free regions become separated by regions that are in use, causing a program to stall, unable to get memory, even though there is theoretically enough available. Memory allocation algorithms that slowly accumulate fragmentation may work perfectly well for desktop machines rebooted every day or so but are unacceptable for embedded systems that often run for months without rebooting.

Memory management is the process by which a computer control system allocates a limited amount of physical memory among its various processes (or tasks) in a way that optimizes performance. Actually, each process has its own private address space, initially divided into three logical segments: text, data, and stack. The text segment is read-only and contains the machine instructions of a program, the data and stack segments are both readable and writable. The data segment contains the initialized and non-initialized data portions of a program, whereas the stack segment holds the application's run-time stack. On most machines, this is extended automatically by the kernel as the process executes. This is done by making a system call, but change to the size of a text segment only happens when its contents are overlaid with data from the file system, or when debugging takes place. The initial contents of the segments of a child process are duplicates of the segments of its parent.

The contents of a process address space do not need to be completely in place for a process to execute. If a process references a part of its address space that is not resident in main memory, the system pages the necessary information into memory. When system resources are scarce, the system uses a two-level approach to maintain available resources. If a modest amount of memory is available, the system will take memory resources away from processes if these resources have not been used recently. Should there be a severe resource shortage, the system will resort to swapping the entire context of a process to secondary storage. This paging and swapping done by the system are effectively transparent to processes, but a process may advise the system about expected future memory utili- zation as a performance aid. A common technique for doing the above is virtual memory, which simulates a much larger address space than is actually available, using a reserved disk area for objects that are not in physical memory. The operating system's kernel often performs memory allocations that are needed for only the duration of a single system call. In a user

process, such short-term memory would be allocated on the run-time stack. Because the kernel has a limited run-time stack, it is not feasible to allocate even moderately- sized blocks of memory on it, so a more dynamic mechanism is needed. For example, when the system must translate a path name, it must allocate a 1-kbyte buffer to hold the name. Other blocks of memory must be more persistent than a single system call, and thus could not be allocated on the stack even if there was space. An example is protocol-control blocks that remain throughout the duration of a network connection.

This section discusses virtual memory techniques, memory allocation and deallocation, memory protection and memory access control.

Virtual memory

When it is executing a program, the microprocessor reads an instruction from memory and decodes it. At this point it may need to fetch or store the contents of a location in memory, so it executes the instruction and then moves on to the next. In this way the microprocessor is always accessing memory, either to fetch instructions or to fetch and store data. In a virtual memory system all of these addresses are virtual, and not physical addresses. They are converted into physical addresses by the microprocessor, based on information held in a set of tables maintained by the operating system.

The operating system uses virtual memory to manage the memory requirements of its processes by combining physical memory with secondary memory (swap space) on a disk, usually located on a hardware disk drive. Diskless systems use a page server to maintain their swap areas on the local disk (extended memory). The translation from virtual to physical addresses is implemented by a memory management unit (MMU), which may be either a module of the CPU, or an auxiliary, closely coupled chip. The operating system is responsible for deciding which parts of the program's simulated main memory are kept in physical memory, and also maintains the translation tables that map between virtual and physical addresses. Three techniques of implementing virtual memory; paging, swapping and segmentation.

(1) Paging

Almost all implementations of virtual memory divide the virtual address space of an application program into pages; a page is a block of contiguous virtual memory addresses. Here, the low-order bits of the binary representation of the virtual address are preserved, and used directly as the low-order bits of the actual physical address; the high-order bits are treated as a key to one or more address translation

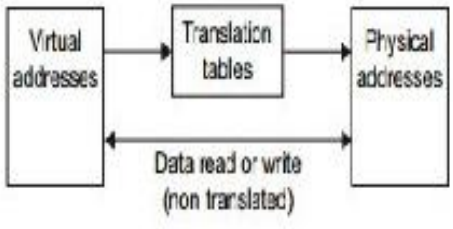


Fig: 4 .10 :Address translation between physical and virtual memory

tables, which provide the high-order bits with the actual physical address. For this reason, a range of consecutive addresses in the virtual address space, whose size is a power of two, will be translated to a corresponding range of consecutive physical addresses. The memory referenced by such a range is called a page. The page size is typically in the range 512 8192 bytes (with 4 kB currently being very common), though 4 MB or even larger may be used for special purposes. (Using the same or a related mechanism, contiguous regions of virtual memory larger than a page are often mappable to contiguous physical memory for purposes other than virtualization, such as setting access and caching control bits.)

Almost all implementations use page tables to translate the virtual addresses seen by the application into physical addresses (also referred to as real addresses) used by the hardware. The operating system stores the address translation tables, i.e. the mappings from virtual to physical page numbers, in a data structure known as a page table. When the CPU tries to reference a memory location that is marked as unavailable, the MMU responds by raising an exception (commonly called a page fault) with the CPU, which then jumps to a routine in the operating system. If the page is in the swap area, this routine invokes an operation called a page swap, to bring in the required page.

The operating systems can have one page table or a separate page table for each application. If there is only one, different applications running at the same time will share a single virtual address space, i.e. they use different parts of a single range of virtual addresses. The operating systems which use multiple page tables provide multiple virtual address spaces, so concurrent applications seem to use the same range of virtual addresses, but their separate page tables redirect to different real addresses

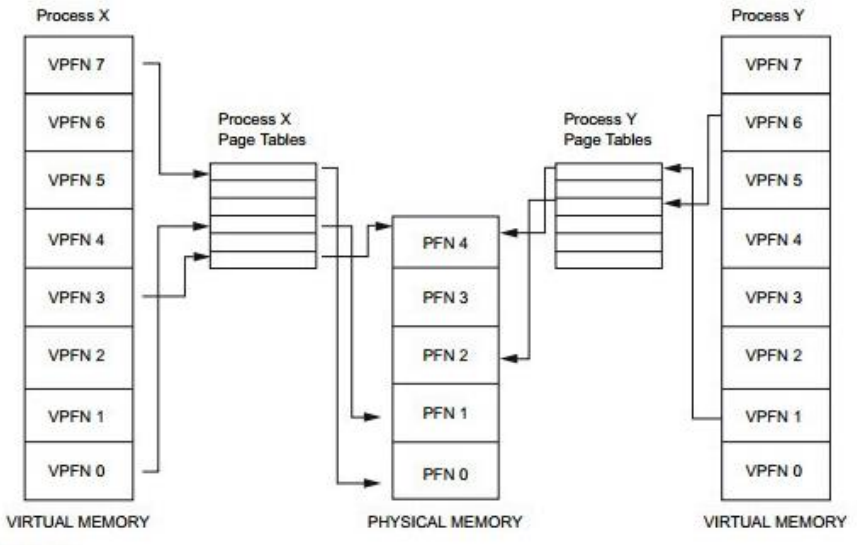


Fig:4.11: Abstract model for mapping virtual addresses to physical addresses in the implementation of virtual memory

The above figure shows the virtual address spaces of two processes, X and Y, each with their own page tables, which map each process's virtual pages into physical pages in memory. This shows that process X's virtual page frame number 0 is mapped into memory at physical page frame number 1 and that process Y's virtual page frame number 1 is mapped into physical page frame number 4. Each entry in the theoretical page table contains the following information: (1) a valid flag that indicates whether this page table entry is valid; (2) the physical page frame number that this entry describes; (3) the access control information that describes how the page may be used.

(2) Swapping

Swap space is a portion of hard disk used for virtual memory that is usually a dedicated partition (i.e., a logically independent section of a hard disk drive), created during the installation of the operating system. Such a partition is also referred to as a swap partition. However, swap space can also be a special file. Although it is generally preferable to use a swap partition rather than a file, sometimes it is not practical to add or expand a partition when the amount of RAM is being increased. In such case, a new swap file can be created with a system call to mark a swap space.
It is also possible for a virtual page to be marked as unavailable because the page was never previously allocated. In such cases, a page of physical memory is allocated and filled with zeros, the page table is modified to describe it, and the program is restarted as above

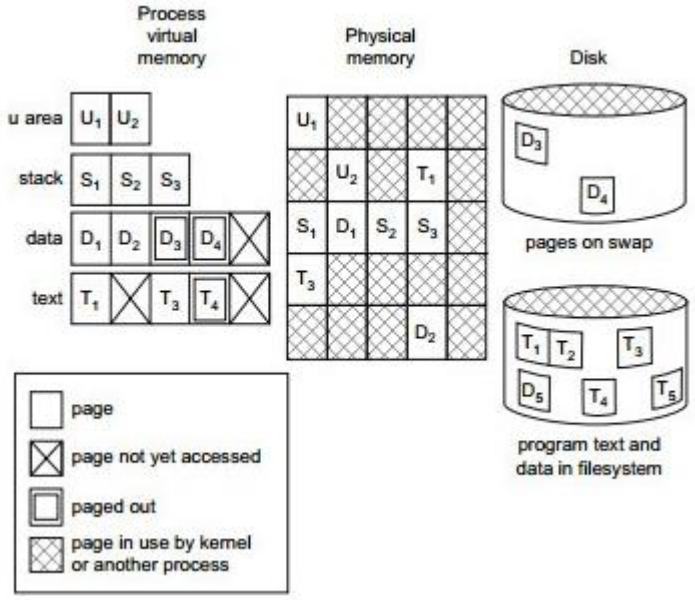


Fig:4.12: Page swapping for implementing virtual memory

The above figure illustrates how the virtual memory of a process might correspond to what exists in physical memory, on swap, and in the file system. The U-area of a process consists of two 4 kB pages (displayed here as U1 and U1) of virtual memory containing information about the process that is needed by the system during execution. In this example, these pages are shown in physical memory, and the data pages, D3 and D4, are shown as being paged out to the swap area on disk. The text page, T4, has also been paged out, but it is

not written to the swap area as it exists in the file system. Those pages that have not yet been accessed by the process (D5, T2, and T5) do not occupy any resources in physical memory or in the swap area. The page swap operation involves a series of steps. Firstly it selects a page in memory; for example, a page that has not been recently accessed and (preferably) has not been modified since it was last read. If the page has been modified, the process writes the modified page to the swap area. The next step in the process is to read in the information in the needed page (the page corre-sponding to the virtual address the original program was trying to reference when the exception occurred) from the swap file. When the page has been read in, the tables for translating virtual addresses to physical addresses are updated to reflect the revised contents of physical memory. Once the page swap completes, it exits, the program is restarted and returns to the point that caused the exception.

(3) Segmentation

Some operating systems do not use paging to implement virtual memory, but use segmentation instead. For an application process, segmentation divides its virtual address space into variable-length segments, so a virtual address consists of a segment number and an offset within the segment. Memory is always physically addressed with a single number (called absolute or linear address). To obtain it, the microprocessor looks up the segment number in a table to find a segment descriptor. This contains a flag indicating whether the segment is present in main memory and, if so, the address of its starting point (segment's base address) and its length. It checks whether the offset within the segment is less than the length of the segment and, if not, generates an interrupt. If a segment is not present in main memory, a hardware interrupt is raised to the operating sys- tem, which may try to read the segment into main memory, or to swap it in. The operating system may need to remove other segments (swap out) in order to make space for the segment to be read in.

The difference between virtual memory implementations that use pages and those using segments is not only about the memory division. Sometimes the segmentation is actually visible to the user processes, as part of the semantics of the memory model. In other words, instead of a process just having a memory which looks like a single large vector of bytes or words, it is more structured. This is different from using pages, which does not change the model visible to the process. This has important consequences. It is possible to combine segmentation and paging, usually by dividing each segment into pages. In such systems, virtual memory is usually implemented by paging, with segmentation used to provide memory protection. The segments reside in a 32-bit linear paged address space, which segments can be moved into and out of, and pages in that linear address space can be moved in and out of main memory, providing two levels of virtual memory. This is quite rare, however, most systems only use paging.

Memory allocation and deallocation

The inefficient allocation or deallocation of memory can be detrimental to system performance. The presence of wasted memory in a block is called internal fragmentation, and it occurs because the size that was requested was smaller than that allocated. The result is a block of unusable memory, which is considered as allocated when not being used. The reverse situation is called external fragmentation, when blocks of memory are freed, leaving non-

contiguous holes. If these holes are not large, they may not be usable because further requests for memory may call for larger blocks. Both internal and external fragmentation results in unusable memory. Memory allocation and deallocation is a process that has several layers of application. If one application fails, another operates to attempt to resolve the request. This whole process is called dynamic memory management. Memory allocation is controlled by a subsystem called malloc, which controls the heap, a region of memory to which memory allocation and deallocation occurs. The reallocation of memory is also under the control of malloc. In malloc, the allocation of memory is performed by two subroutines, malloc and calloc. Deallocation is performed by the free subroutine, and reallocation is performed by the subroutine known as realloc. In deallocation, those memory blocks that have been deallocated are returned to the binary tree at its base. Thus, a binary tree can be envisioned as a sort of river of information, with deallocated memory flowing in at the base and allocated memory flowing out from the tips.

Garbage collection is another term associated with deallocation of memory. This refers to an automated process that determines what memory is no longer in use, and so recycles it. The automation of garbage collection relieves the user of time-consuming and error-prone tasks. There are a number of algorithms for the garbage collection process, all of which operate independently of malloc.

Memory allocation and deallocation can be categorized as static or dynamic.

(1) Static memory allocation

Static memory allocation refers to the process of allocating memory at compile-time, before execution. One way to use this technique involves a program module (e.g., function or subroutine) declaring static data locally, such that these data are inaccessible to other modules unless references to them are passed as parameters or returned. A single copy of this static data is retained and is accessible through many calls to the function in which it is declared. Static memory allocation therefore has the advantage of modularizing data within a program so that it can be used throughout run-time. The use of static variables within a class in object-oriented programming creates a single copy of such data to be shared among all the objects of that class.

(2) Dynamic memory allocation

Dynamic memory allocation is the allocation of memory storage for use during the run-time of a program, and is a way of distributing ownership of limited memory resources among many pieces of data and code. A dynamically allocated object remains allocated until it is deallocated explicitly, either by the programmer or by a garbage collector; this is notably different from automatic and static memory allocation. It is said that such an object has dynamic lifetime. Memory pools allow dynamic memory allocation comparable to malloc, or the operator "new". As those implementations suffer from fragmentation because of variable block sizes, it can be impossible to use them in a real-time system due to performance problems.

A more efficient solution is to pre-allocate a number of memory blocks of the same size, called the memory pool. The application can allocate, access, and free blocks represented by handles at run- time. Fulfilling an allocation request, which involves finding a block of unused memory of a certain size in the heap, is a difficult problem. A wide variety of solutions have been proposed, and some of the most commonly used are discussed here.

(a) Free lists

A free list is a data structure used in a scheme for dynamic memory allocation that operates by connecting unallocated regions of memory together in a linked list, using the first word of each unallocated region as a pointer to the next. It is most suitable for allocating from a memory pool, where all objects have the same size. Free lists make the allocation and deallocation operations very simple. To free a region, it is just added it to the free list. To allocate a region, we simply remove a single region from the end of the free list and use it. If the regions are variable-sized, we may have to search for a large enough region, which can be expensive. Free lists have the disadvantage, inherited from linked lists, of poor locality of reference and thus poor data cache utilization, and they provide no way of consolidating adjacent regions to fulfill allocation requests for large regions. Nevertheless, they are still useful in a variety of simple applications where a full-blown memory allocator is unnecessary, or requires too much overhead.

(b) Paging

As mentioned earlier, the memory access part of paging is done at the hardware level through page tables, and is handled by the MMU. Physical memory is divided into small blocks called pages (typically 4 kB or less in size), and each block is assigned a page number. The operating system may keep a list of free pages in its memory, or may choose to probe the memory each time a request is made (though most modern operating systems do the former). In either case, when a program makes a request for memory, the operating system allocates a number of pages to it, and keeps a list of allocated pages for that particular program in memory.

Memory protection

The topic of memory management in this section addresses a different set of constructs related to physical and virtual memory: protected memory, infinite amount of memory, and transparent sharing. Perhaps the simplest model for using memory is to provide single programming without memory protection, where each process (or task) runs with a range of physical memory addresses. Given that a single-programming environment allows only one process to run at a time, this can use the same physical addresses every time, even across reboots. Typically, processes use the lower memory addresses (low memory), and an operating system uses the higher memory addresses (high memory). An application process can address any physical memory location.

One step beyond the single-programming model is to provide multiprogramming without memory protection. When a program is copied into memory, a linker-loader alters the code of the program (loads, stores, jumps) to use the address of where the program lands in memory. In this environment, bugs in any program can cause other programs to crash, even the operating system. The third model is to have a multitasking operating system with memory protection, which keeps user programs from crashing one another and the operating system. Typically, this is achieved by two hardware- supported mechanisms: address translation and dual-mode operation.

(1) Address translation

Each process is associated with an address space, or all the physical addresses it can touch. However, each process appears to own the entire memory, with the starting virtual address of 0. The missing piece is a translation table that translates every memory reference from virtual addresses to physical addresses. Translation provides protection because there is no way for a process to talk about other processes' address, and it has no way of touching the code or data of the operating system. The operating system uses physical addresses directly, and involves no translation. When an exception occurs, the operating system is responsible for allocating an area of physical memory to hold the missing information (and possibly in the process pushing something else out to disk), bringing the relevant information in from the disk, updating the translation tables, and finally resuming execution of the software that incurred the exception.

(2) Dual-mode operation

Translation tables can offer protection only if a process cannot alter their content. Therefore, a user process is restricted to only touching its address space under the user mode. A CPU can change from kernel to user mode when starting a program, or vice versa through either voluntary or involuntary mechanisms. The voluntary mechanism uses system calls, where a user application asks the operating system to do something on its behalf. A system call passes arguments to an operating system, either through registers or copying from the user memory to the kernel memory. A CPU can also be switched from user to kernel mode involuntarily by hardware interrupts (e.g., I/O) and program exceptions (e.g., segmentation fault).

On system calls, interrupts, or exceptions, hardware atomically performs the following steps:
(1) sets the CPU to kernel mode; (2) saves the current program counter; (3) jumps to the handler in the kernel (the handler saves old register values).
Unlike threads, context switching among processes also involves saving and restoring pointers to translation tables. To resume execution, the kernel reloads old register values, sets the CPU to user mode, and jumps to the old program counter. Communication among address spaces is required in this operation. Since address spaces do not share memory, processes have to perform inter-process communication (IPC) through the kernel, which can allow bugs to propagate from one program to another. Protection by hardware can be prohibitively slow, since applications have to be structured to run in separate address spaces to achieve fault isolation. In the case of complex applications built by multiple vendors, it

may be desirable for two different programs to run in the same address space, with guarantees that they cannot trash each other's code or data. Strong typing and software fault isolation are used to ensure this.

(a) Protection via strong typing

If a programming language disallows the misuse of data structures, a program may trash another, even in the same address space. With some object-oriented programming, programs can be downloaded over the net and run safely because the language, compiler, and run-time system prevents the program from doing bad things (e.g., make system calls). For example, Java defines a separate virtual machine layer, so a Java program can run on different hardware and operating systems. The downside of this protection mechanism is the requirement to learn a new language.

(b) Protection via software fault isolation

A language-independent approach is to have compilers generate code that is proven safe (e.g., a pointer cannot reference illegal addresses). For example, a pointer can be checked before it is used in some applications.

Memory access control

Dealing with race conditions is also one of the difficult aspects of memory management. To manage memory access requests coming from the system, a scheduler is necessary in the application layer or in the kernel, in addition to the MMU as a hardware manager. The most common way of protecting data from concurrent access by the memory access request scheduler is memory request contention. The semantics and methodologies of memory access request contention should be the same as for I/O request contention.

Timer Management

Tasks need to be performed after scheduled durations. To keep track of the delays, timers- relative and absolute are provided in RTOS.

Interrupt and event handling

RTOS provides various functions for interrupt and event handling, viz., Defining interrupt handler, creation and deletion of ISR, referencing the state of an ISR, enabling and disabling of an interrupt, etc. It also restricts interrupts from occurring when modifying a data structure, minimize interrupt latencies due to disabling of interrupts when RTOS is performing critical operations, minimizes interrupt response times.

Device I/O Management

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc. An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories −

- **Block devices − A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.**
- **Character devices − A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc**

Device Controllers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller. There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.

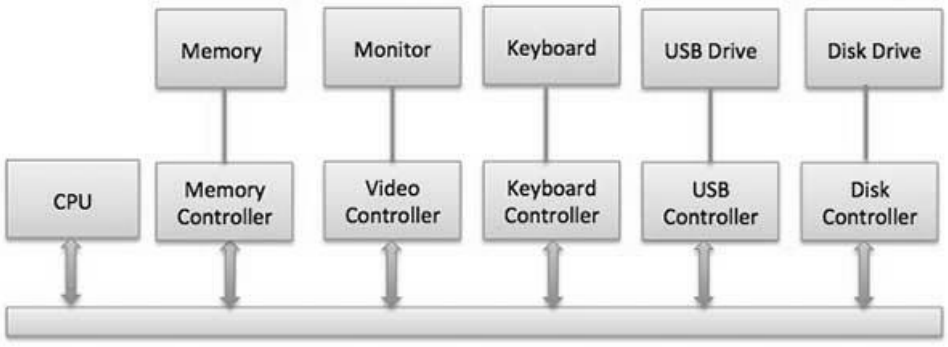


Fig:4.13: Model for connecting the CPU to peripherals

Synchronous vs asynchronous I/O

- **Synchronous I/O − In this scheme CPU execution waits while I/O proceeds**
- **Asynchronous I/O − I/O proceeds concurrently with CPU execution**

Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

- **Special Instruction I/O**
- **Memory-mapped I/O**
- **Direct memory access (DMA)**

Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.

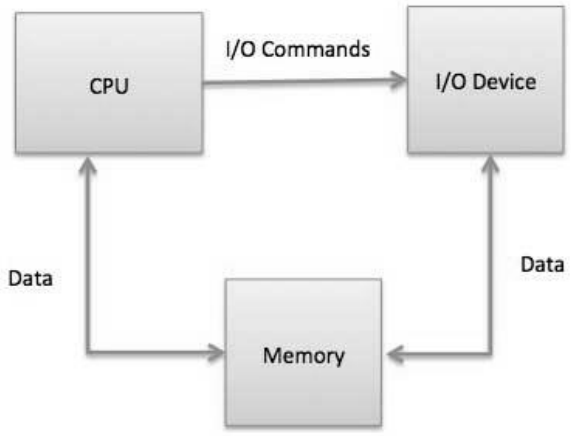


Fig:4.14: Memory-mapped I/O

While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished. The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

Direct Memory Access (DMA)

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead. Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main

memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred. Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.

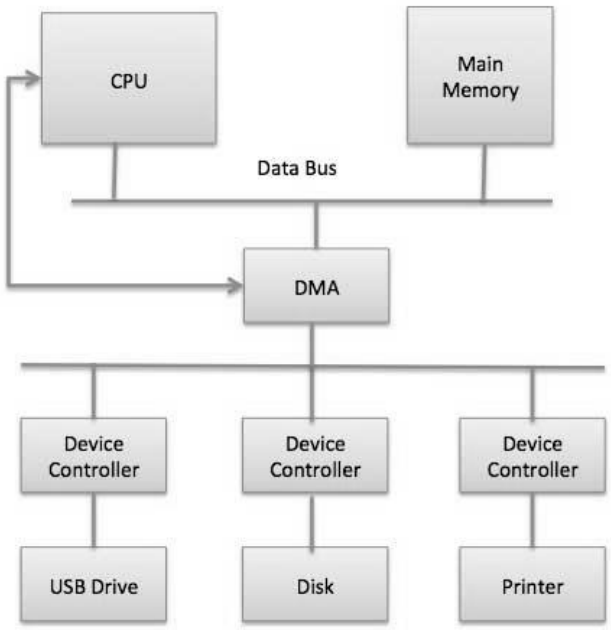


Fig:4.15: Direct Memory Access

I/O software is often organized in the following layers −

- **User Level Libraries − This provides simple interface to the user program to perform input and output. For example, stdio is a library provided by C and C++ programming languages.**
- **Kernel Level Modules − This provides device driver to interact with the device controller and device independent I/O modules used by the device drivers.**
- **Hardware − This layer includes actual hardware and hardware controller which interact with the device drivers and makes hardware alive.**

A key concept in the design of I/O software is that it should be device independent where it should be possible to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.

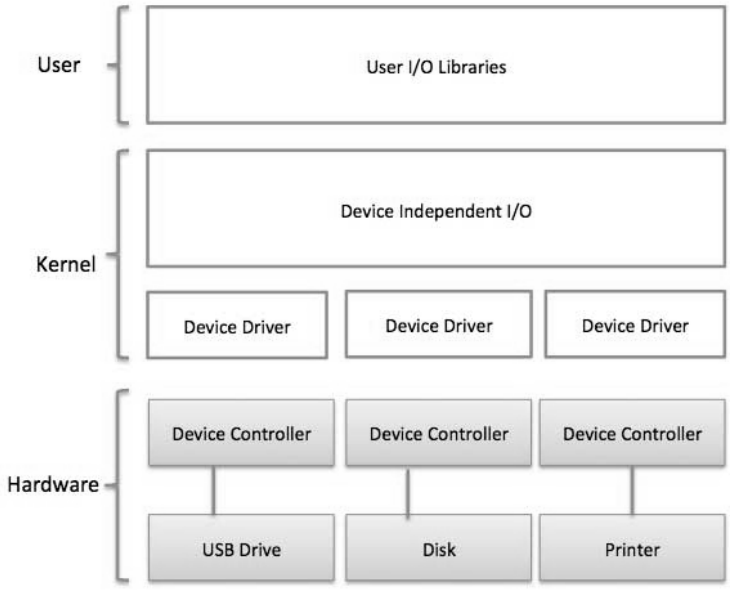


Fig:4.16: Design of I/O software

Device Drivers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. Device drivers encapsulate device-dependent code and implement a standard interface in such a way that code contains device-specific register reads/writes. Device driver, is generally written by the device's manufacturer and delivered along with the device on a CD-ROM.

A device driver performs the following jobs −

- **To accept request from the device independent software above to it.**
- **Interact with the device controller to take and give I/O and perform required error handling**
- **Making sure that the request is executed successfully**

How a device driver handles a request is as follows: Suppose a request comes to read a block N. If the driver is idle at the time a request arrives, it starts carrying out the request immediately. Otherwise, if the driver is already busy with some other request, it places the new request in the queue of pending requests.

Interrupt handlers

An interrupt handler, also known as an interrupt service routine or ISR, is a piece of software or more specifically a callback function in an operating system or more specifically in a device driver, whose execution is triggered by the reception of an interrupt. When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt, updates data structures and wakes up process that was waiting for an interrupt to happen.

The interrupt mechanism accepts an address ─ a number that selects a specific interrupt handling routine/function from a small set. In most architectures, this address is an offset stored in a table called the interrupt vector table. This vector contains the memory addresses of specialized interrupt handlers.

Device-Independent I/O Software

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Though it is difficult to write completely device independent software but we can write some modules which are common among all the devices. Following is a list of functions of device-independent I/O Software −

- **Uniform interfacing for device drivers**
- **Device naming - Mnemonic names mapped to Major and Minor device numbers**
- **Device protection**
- **Providing a device-independent block size**
- **Buffering because data coming off a device cannot be stored in final destination.**
- **Storage allocation on block devices**
- **Allocation and releasing dedicated devices**
- **Error Reporting**

User-Space I/O Software

These are the libraries which provide richer and simplified interface to access the functionality of the kernel or ultimately interactive with the device drivers. Most of the user-level I/O software consists of library procedures with some exception like spooling system which is a way of dealing with dedicated I/O devices in a multiprogramming system. I/O Libraries (e.g., stdio) are in user-space to provide an interface to the OS resident device-independent I/O SW. For example putchar(), getchar(), printf() and scanf() are example of user level I/O library stdio available in C programming. RTOS generally provides large number of APIs to support diverse hardware device drivers.

TEXT / REFERENCE BOOKS

1. Jane W. S Liu, "Real Time Systems" Pearson Higher Education, 3rd Edition, 2000.
2. Raj Kamal, "Embedded Systems- Architecture, Programming and Design" Tata McGraw Hill, 2nd Edition, 2014.
3. Jean J. Labrosse, "Micro C/OS-I : The real time kernel" CMP Books, 2nd Edition,2015.
5. Richard Barry, "Mastering the Free RTOS: Real Time Kernel", Real Time Engineers Ltd, 1st Edition, 2016.
6. David E. Simon, " An Embedded Software Primer", Pearson Education Asia Publication

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL & ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT -V

RTOS PROGRAMMING - SECA5302

V RTOS IMAGE BUILDING FOR DIFFERENT TARGET PLATFORMS

Porting of RTOS, Configuring RTOS for minimizing RAM consumption and increasing Throughput- Building RTOS Image for Target platforms

Porting of RTOS:

Product development cycles are market driven, and market demands often require vendors to compress development schedules. One approach to this is to simultaneously develop similar products, yet with varying levels of product complexity. However, scheduling pressures coupled with increased product complexity can be a recipe for disaster, resulting in slipped schedules and missed opportunities. Consequently, vendors are always on the alert for silver bullets, yet as developers, we know that they don't exist. That said, it is still in our best interest to seek better ways of compressing development cycles, and one way to do this is to port existing products to new hardware platforms, adding new features along the way. This is the approach we used to demonstrate a proof-in-concept when porting a legacy security application to a new hardware platform.

Our firm was hired to make enhancements to the client's existing 6502-based product, and we quickly realized that this platform was running out of steam. Specifically, the proposed features would significantly impact performance. Consequently, we proposed three options for fixing this problem:

- **Completely rewriting the application on the current hardware.**
- **Rewriting the application on a new, higher performance hardware.**
- **Migrating portable portions of the application to the new hardware. After considering the options, we decided to port to new hardware.**

RTXC Overview

The Real-Time executive kernel (RTXC) supports three kinds of priority- based task scheduling: preemptive (the default), round-robin, and time-slice. RTXC is robust, supports hard deadlines, changeable task priorities, time and resource management, and inter task communication. It also has a small RAM/ROM code footprint, standard API interface, and is implemented in many processors. RTXC is divided into nine basic components: tasks, mailboxes, messages, queues, semaphores, resources, memory partitions, timers, and Interrupt Service Routines (ISRs). These components are further subdivided into three groups that are used for inter task communication, synchronization, and resource management. Moreover, component functionality is accessed via the standard API interface.

Porting Activities Overview:

Porting an RTOS to a new board requires four activities:

- **Determining the system's architecture.**
- **Figuring out what files to change based on the architecture.**
- **Making changes to the files; this includes writing the code.**
- **Creating test code and exercising the board to ensure that the RTOS is working properly.**

The first activity is design related, while the others are implementation related. Moreover, the last three activities require an understanding of the new hardware—knowing the specifics of what needs to happen to make the RTOS interact with the board.

System Architecture:

The purpose of determining the system architecture requirements is to identify the hardware and software components that need modifying to get the RTOS up and running on the NPE-167 board. For most porting projects, hardware components include I/O, memory, timers, and other unique peripherals. For this project, these components are no different. We had the I/O ports that control the LEDs, CAN bus, serial communication, memory selection, and card-slot selection. Memory had both Flash and SRAM. Memory is selected through the I/O component using the SPI bus. Therefore, I/O and memory selection are interrelated. For this project, we also had to identify the timer to run RTXC's real- time clock. The real-time clock is the master timer used for all RTOS-based time-keeping functions. Additionally, for this project, we were not going to use any other on-chip peripherals.

The best way to identify hardware components is to study the board's schematics. Examining the NPE-167 board revealed that the I/O ports would be key for this project. Why? Because this board used the processor's general-purpose ports to handle switches to control CAN bus operation, the board's operating mode, control LED outputs, and memory selection. I/O cards were controlled via the SPI bus, rather than I/O ports.

Ports can as either inputs or outputs. Examination of the NPE- 167 board showed that 17 be configured ports are used. Eleven ports are used as switch inputs. From the schematic we saw that switches 1-7 were used to set the MAC address for the CAN device. CAN bus speed is controlled by switches 8-9, while the board operating mode is controlled by switches 11-12. Switch 10 is not used. Four ports control the LEDs. There are three in total. One LED is green, one red, and the third bicolor. Thus, four outputs are required to control the three LEDs. Finally, two output ports are used as page selection for extended memory.

NPE board addresses up to 512K of memory before having to make use of the page-

selection ports. Although we would configure the page-selection ports for the porting process, we didn't have to use them because the total code footprint of the kernel, plus test code, is 107K. RTXC's kernel is about 76K, and the porting test code fits within another 31K. In short, we would only use about 1/5 of the default memory to validate the porting process.

The last necessary component for the port was to determine which timer to use as the master time base. Timers are internal on the C167 processor, so they don't show up on the schematic. So we had two options—choose a timer and write the code for that timer, or use the BSP default timer. RTXC's C167 BSP uses a timer in its configuration. A trick to simplify the initial porting process is to use the default timer that the BSP uses. Reviewing the BSP documentation, we discovered that it uses timer 6 for the master timer. Once we determined the components associated with the porting process, we could turn our attention to figuring out which files needed to be changed.

Changing Files

We knew from the previous step that 11 ports were used for input and six ports for output. Because these were general-purpose I/O ports, they needed to be initialized to work as either inputs or outputs. This gave us an idea of where NPE- specific initialization code needed to go—specifically, initialization code to set up these ports goes in the startup code. For this project, initialization code was located in the cstart.a66 file that is located in the Porting directory. Listing One is the code that configures the NPE-167 board I/O. Once configured, I/O can be used by higher level RTOS and API functions. Once we figured out where the I/O changes go, we needed to turn our attention to discovering and setting up the master timer.

BSP set up the master timer for us because we were using default timer 6. Setup code for this timer is located in cstart.a66 and rtxc main.c. Listing Two is a snippet of the RTXC-specific code. After analyzing the architecture requirements, we discovered that the only file to change for porting the NPE-167 board was cstart.a66. Granted, we knew we would have to change other files as well, but those files are application specific.

Changing Files and Writing Code

This brought us to the third step, which was straightforward because we knew what needed to be changed and where. Recall that all changes for basic porting functionality occurred in cstart.a66. We also needed to write the code for initialization. We wrote code to initialize the switches to handle CAN—but no other code— to deal with it because it is not used in the basic port. For specifics, look at cstart.a66 and search for npe and rtxc labels to find code changes specific to this port. Keep in mind, when porting to new hardware you may want to adopt a similar strategy for partitioning the code for hardware- and RTOS-specific changes. That is because partitioning code through the use of labels helps with code maintainability.

Test Code

Finally, we needed to create some test code to test our port. Building the test code application was a two-step process:

- **We compiled the RTXC kernel into a library object (rtxc.lib).**
- **We compiled the test code and link in rtxc.lib to create the executable.**

There are two directories for generating the test code, and they are stored at the same level in the hierarchy. Moreover, all files for creating rtxc.lib are located in the kernel directory. Alternatively, test code-specific files are located in the Porting directory.

The RTXCgen utility creates a set of files corresponding to each RTOS component. For instance, application queues are defined in three files: cqueue.c, cqueue.h, and cqueue.def. The same holds true for tasks, timers, semaphores, mailboxes, and the rest. Changes to the number of RTOS components are handled by this utility. For example, if we wanted to change the number of tasks used by the test code, we use RTXCgen to do it. Figure 2 shows the contents of the task definition file for the test code application. Test code files created by RTXCgen are placed in the Porting directory. Once RTXCgen has defined the system resources, we are ready to build the project.

Creating the executable test code requires the build of two subprojects—the kernel and test code. We performed builds using the Keil Microvision IDE (http://www.keil.com/). Keil uses project files (*.prj files) to store its build information. RTXC kernel creation consists of building the code using the librtxc.prj file located in the kernel directory. Evoking the librtxc project compiles, links, and creates a librtxc object in the kernel directory. Building the test code is accomplished using the NpeEg.prj file stored in the Porting directory. Invoking the NpeEg project compiles and links files in the Porting directory, and links the librtxc object in the kernel directory. The resulting executable is then placed in the Porting directory as well. Once the test code was fully built, we were ready to test the board port.

The test code is a simple application used to validate the porting process. Most of the test code is located in main.c located in the Porting directory. The application works by starting five tasks—two user and three system. User tasks execute alternatively, while system tasks execute in the background. One user task begins running. It then outputs data via one of the system tasks to the console. Next, it signals the other to wake up, and it puts itself to sleep, thus waiting for the other task to signal it to wake up again.

Most embedded systems of non-trivial complexity employ an operating system of some kind – commonly an RTOS. Ultimately, the OS is an overhead, which uses time and memory that could otherwise have been used by the application code. As an embedded system has limited resources, this overhead needs to be carefully evaluated, which commonly leads to questions about RTOS memory footprint. If you were considering the purchase of a real time operating system (or, for that matter, any piece of software IP for an embedded application), you would probably like to get clear information on the amount of memory that it uses. It is very likely that an RTOS vendor will be unwilling – or actually, to be more precise – unable to provide you with such seemingly obvious information. The reason for this is that there are a huge number of variables.

Broadly speaking, there is read only memory (ROM – nowadays that is usually flash

memory) and read/write memory (RAM). ROM is where the code and constant data is stored; RAM is used for variables. However, to improve performance, it is not uncommon to copy code/data from ROM to RAM on boot up and then use the RAM copy. This is effective because RAM is normally faster to access than ROM. So, when thinking about of RTOS footprint, you need to consider ROM and RAM size, including the RAM copy possibility.

The issue can become more complex. There may be on-chip RAM and external memory available. The on-chip storage is likely to be faster, so it may be advantageous to ensure that RTOS code/data is stored there, as its performance will affect the whole application. In a similar fashion, code/data may be locked into cache memory, which tends to offer even higher performance.

Compiler optimization

When building code, like an RTOS, the optimization setting applied to the compiler affect both size and execution speed. Most of the time, code built for highest performance (i.e. fastest) will be bigger; code optimized to be smaller will run slower. It is most likely that an RTOS would normally be built for performance, not size.
Although an RTOS vendor, wanting to emphasize the small size of their product, might make a different choice.

RTOS configuration

Real time operating systems tend to be very configurable and that configuration can vary the RTOS size drastically. Most RTOS products are scalable, so the memory footprint is determined by the actual services used by the application. The granularity of such scalability varies from one product to another. In some cases, each individual service is optional; in others, whole service groups are included or excluded – i.e. if support for a particular type of RTOS object (e.g. semaphore) is required, all the relevant services are included. On a larger scale, other options, like graphics, networking and other connectivity, will affect the code size, as these options may or may not be needed/included.

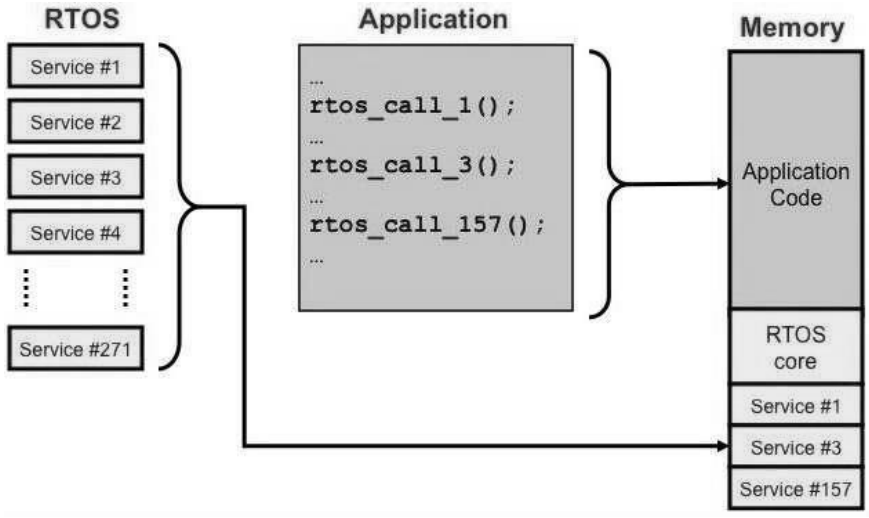


Fig:5.1:RTOS Configuration

Runtime library

Typically, a runtime library will be used alongside an RTOS; this code needs to be accommodated. Again, the code, being a library, may scale well according to the needs of a particular application. Data size issues apart from a baseline amount of storage for variables, the RAM requirements of an RTOS can similarly be affected by a number of factors:

Compiler optimization

As with code, compiler optimization affects data size. Packed (compressed)data is smaller, but takes more instructions, and hence more time, to access.

RTOS objects

The number of RTOS objects (tasks, mailboxes, semaphores etc.) used by the application will affect the RTOSRAM usage, as each object needs some RAM space.

Stack

Normally, the operating system has a stack and every task has its own stack; these must all be stored in RAM. Allocation of this space may be done differently in each RTOS, but it can never be ignored.

Dynamic memory

If dynamic (partition/block)memory allocation is available with an RTOS and used by the application, space for memory pools needs to be accommodated.

Static and dynamic RTOS configuration

Early RTOS products required configuration to be performed at build time – i.e. statically. As the technology progressed, the facility to create (and destroy) RTOS objects dynamically became commonplace. It is now quite uncommon to find an RTOS that permits static configuration. The impact on memory utilization of these options is interesting.

A statically configured RTOS holds most the data about RTOS objects in ROM. Some information needs to be copied to RAM, as it will be changed during execution, but needs to be initialized. Other objects need extra RAM space at run time.

A dynamically configured RTOS keeps all object data in RAM – none in ROM at all. However, there is a significant hit on ROM space, as there will be extra service calls to perform object creation and destruction.

RTOS for Image Processing:

Image registration is the process of transforming a set of sequential images (video stream acquired from a sensor) into a similar coordinate system, creating a smoother visual flow. In real life, physical conditions or normal movement affect the images a sensor gathers and may cause vibrations. Viewing a continuous frame-set from an image sensor generally looks shaky or unbalanced, as the sensor is often mobile or not stabilized. Image registration fixes this problem by smoothing the output video stream. Applications for image registration vary from defense to medical imaging and more.

Typical registration process stages include identifying movement vectors between two relative images, performing alignment, and applying further correction/enhancement filters to improve image and stream quality. In defense applications, sensor-based components use registration from ground systems to a variety of aerial systems. Adding to its complexity, defense applications require very high-performance computations (high resolutions and frame rates) and have limited space for hardware, dictating a small system size. This requires a solution with good heat dissipation and the ability to consistently operate at low power.

The key points in RTOS for image processing applications are :

- **The need for speed**
- **Power and size constraints**
- **Real-Time threat detection**
- **Faster-than-real-time processing**

The quality and the size of image data (especially 3D medical Segmentation of a human brain. data) is constantly increasing. Fast and optimally interactive post processing of these images is a major concern. E. g., segmentation on them, morphing of different images, sequence analysis and measurement are difficult tasks to be performed. Especially for segmentation and for morphing purposes level set methods play an important role.

In the case of image segmentation f is a force which pushes the interface towards the boundary of a segment region in an image. Usually f equals one in homogeneity regions of the image, whereas f tends to zero close to the segment boundary. The discretization of the level set model is performed with finite differences on an uniform quadrilateral or octahedral grid. A characteristic of image processing methods is the above described multiple iterative processing of data sets. Due to the possible restriction on the number precision it is possible to work on integer data sets with a restricted range of values, i. e., an application specific word length. Furthermore, it is possible to incorporate parallel execution of the update formulas

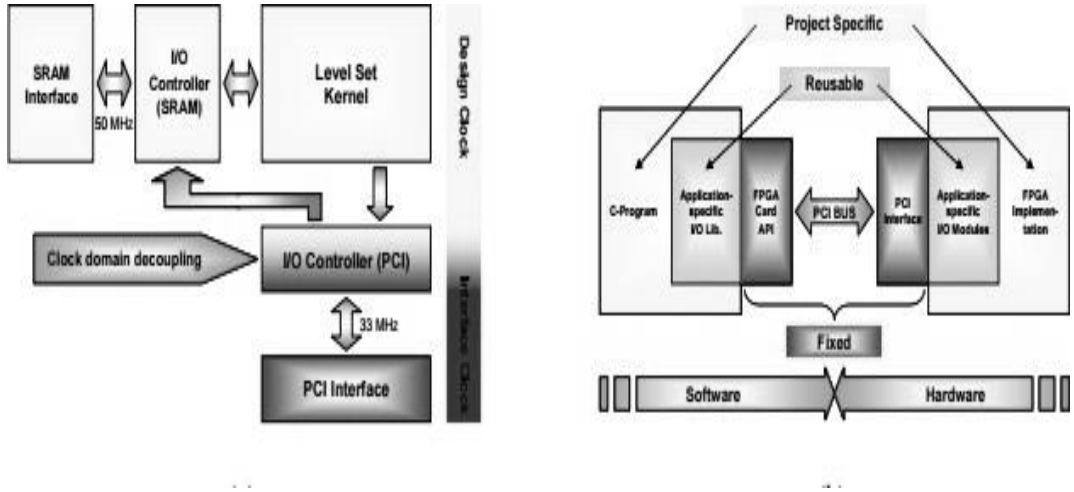


Fig:5.2: A hardware accelerated system for data Processing: Software and Hardware Modules

Image processing algorithms as described consists of a complex sequence of primitive operations, which have to be performed on each nodal value. By combining the complete sequence of primitive operations into a compound operation it is possible to reduce the loss of performance caused by the synchronous design approach.

This design approach, which is common to CPU designs as well as for FPGA designs, is based on the assumption that all arithmetic operations will converge well in advance to the clock tick, which will cause the results to be post processed. Therefore, the maximum clock speed of such systems is defined by the slowest combinatorial path. In a CPU this leads to 'waiting time' for many operations. Furthermore, there is no need for command fetching in FPGA designs, which solves another problem of CPU-based algorithms. Additionally, it is possible to do arbitrary parallel data processing in a FPGA, so that several nodal values can be updated simultaneously.

The input data rate for CPU-based and FPGA-based applications is determined by the bandwidth of the available memory interface. A 2562 image results in 64k words, resulting in 768 kBit data at 12 bit resolution. A CPU with a 16 bit wide data access would need 128 kByte to store the original image data, without taking the memory for intermediate results into account. The discussion is not restricted to the information described in this section, there are many real time examples those can be considered as case studies for image processing in RTOS. The process for generating a target image for the QNX CAR platform is described below

Overall image generation process

The following illustration shows the process used to generate a QNX CAR platform target image:

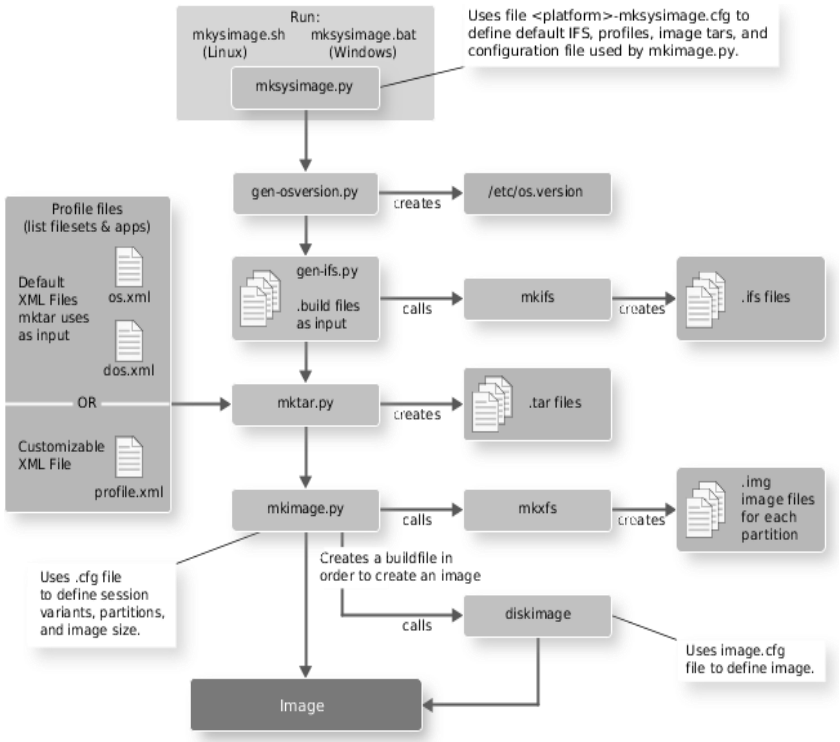


Fig:5.3: Procedure to generate a QNX CAR platform target image

Building the target image:

1. Set up the environment variables for the QNX development environment:

For Linux, enter the following command:

 # source *install_location* /qnx660-env.sh

where by default the *install_location* is $ *HOME* /qnx660/.

For Windows, enter the following command:

***install_location* \qnx660-env.bat**

where by default the *install_location* is C:\qnx660.

As part of the installation process for the QNX CAR platform, a workspace was created for you that contains the scripts and configuration files you'll be using. These files are located in the following
locations:

Scripts:

For Linux: $QNX_CAR_DEPLOYMENT /deployment/scripts/

For Windows: %QNX_CAR_DEPLOYMENT% \deployment\scripts

where QNX_CAR_DEPLOYMENT is install_location /qnx660/deployment/qnx-car/.

Configuration files:

For Linux: $QNX_CAR_DEPLOYMENT /boards/<platform >/etc/

For Windows: %QNX_CAR_DEPLOYMENT% \boards\<platform >\etc

2. Extract a BSP. For detailed instructions, see "*Building a BSP* ".

3. Create an output directory where you want to have the image generated.

You must specify a valid directory name; the directory must exist prior to running the mksysimage.py script, otherwise the image won't be generated.

4. To generate a target system image, run the appropriate mksysimage.py command.

For Linux, enter the following command:

mksysimage.sh *outputPath platform* .external

For Windows, enter the following command:

mksysimage.bat *outputPath platform* .external

where *outputPath* is the location for the new system image. If this directory isn't empty, run

mksysimage.py with the -f option (mksysimage.py won't overwrite existing system images

The mksysimage.py utility generates images for various configurations. For example, for SABRE

Lite, image files are created for SD and SD/SATA:

imx61sabre-dos-sd-sata.tar
imx61sabre-dos-sd.tar
imx61sabre-os.tar
imx61sabre-sd-sata.img
imx61sabre-sd.img

TEXT / REFERENCE BOOKS

1. Jane W. S Liu, "Real Time Systems" Pearson Higher Education, 3rd Edition, 2000.
2. Raj Kamal, "Embedded Systems- Architecture, Programming and Design" Tata McGraw Hill, 2nd Edition, 2014.
3. Jean J. Labrosse, "Micro C/OS-I : The real time kernel" CMP Books, 2nd Edition,2015.
5. Richard Barry, "Mastering the Free RTOS: Real Time Kernel", Real Time Engineers Ltd, 1st Edition, 2016.
6. David E. Simon, " An Embedded Software Primer", Pearson Education Asia Publication