Time Management in Partitioned Systems

A Thesis

Submitted for the Degree of

Master of Science (Engineering)

in the Faculty of Engineering

by

A. Hariprasad Kodancha

Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012

October 2007

Submitted to the lotus feet of

Lord Shri Krishna

Acknowledgements

I sincerely thank my advisor Prof K. Gopinath for his guidance and support to complete this thesis. I also thank my organization, Accord Software and Systems for supporting me with time and resources to complete my thesis. I thank my organizational guide Mr.Purushotham for his whole-hearted support. I would also like to thank my supervisors Mr.Devanathan, Mr.Shashidhar and Mr.Rengasamy for their encouragement and support. I will always remember the sincere help that I received from my colleagues Prashanth Nayak and Yadunandan. I believe this has been possible due to the continued motivation and support from my wife.

Abstract

Time management is one of the critical modules of safety-critical systems. Applications need strong assurance from the operating system that their hard real-time requirements are met. Partitioned system has recently evolved as a means to provide protection to safety critical applications running on an Avionics computer resource. Each partition has an application running strictly for a specified duration. These applications use the CPU on a cyclic basis. Applications running on a real-time systems request the service of time management in one way or the other. An application may request for a time-out while waiting for a resource, may voluntarily relinquish the CPU for some delay time or may have deadline before which it is expected to complete its tasks. These requests must be handled in a deterministic and accurate way with lower overheads. Time management within an operating system uses the hardware timers to service the time-out requests.

The three well-known approaches for handling timer requests are tick-based, one-shot and firm timer. Traditionally tick-based has been the most popular approach that relies on periodic interrupt timer, although it has a poor accuracy. One-shot timer approach provides better accuracy as the timer interrupt can be generated exactly when required. Firm timers use soft timers in combination with one-shot timer wherein the expired timers are checked at strategic points in the kernel. The thesis compares the performance of these three approaches for partitioned systems and provides an insight about the suitability of the approaches. It also compares various algorithms of tick-based and one-shot timer. This thesis presents a new one-shot timer algorithm named hierarchical multiple linked lists and the experimental results proves its superiority. The hard real-time system under consideration is avionics system and an indigenously developed ARINC-653 compliant real-time operating system has been used to measure the performance.

Contents

Chapter 1

Introduction

Time management is an important component of the operating system used in real-time systems. The avionics computer resource is an embedded generic computing platform that is able to host multiple applications with different levels of criticality. Safety-critical applications such as flight control, cockpit display, navigation, radar control etc., run on an avionics computer resource. These applications need strong assurance from the operating system that their hard real-time requirements are met. More importantly, they require the assurance that in presence of faults, a fault in one application should not propagate to the others. Computing systems in which consequences of failure are very serious are termed as safety-critical. A key feature of a real-time operating system (RTOS) is its ability to meet tight processing deadlines. An application program controlling the release of a weapon may require an action from the operating system in less than one-thousandth of a second.

1.1 Partitioned System

Partitioned system has recently evolved as a means to provide protection to safety critical applications running on an Avionics computer resource [1]. Every partition has an application running strictly for a specified duration. There could be many such partitions using the CPU on a cyclic basis. Partitioned systems allow applications with different criticalities to co-exist and run on the same core module without causing any potential damages to other partitions or applications. A partitioned system is a static system where the time to start and stop an application is predetermined.

1.1.1 Evolution of partitioned system

Automated aircraft control has traditionally been divided into distinct functions that are implemented separately (e.g., autopilot, auto-throttle, flight management); each function has

its own fault tolerant computer system. A by-product of this federated architecture is that faults are strongly contained within the computer system of the function where they occur and cannot readily propagate to affect the operation of other functions. The obvious disadvantage to the federated approach is its profligate use of resources: each function needs its own computer system (which is generally replicated for fault tolerance), with all the attendant costs of acquisition, space, power, weight, cooling, installation and maintenance.

Integrated Modular Avionics (IMA) has therefore emerged as a design concept with a single computer system (with internal replication to provide fault tolerance) that provides a common computing resource to several functions [2], [10]. As a shared resource, IMA has the potential to diminish fault containment between functions, so any realization of IMA must provide *partitioning* [1] to ensure that the shared computer system provides protection against fault propagation from one function to another. Partitioning uses appropriate hardware and software mechanisms to restore strong fault containment to such integrated architectures. IMA proposes the integration of avionics software functions to save physical resources. Also, to cut development costs, IMA encompasses a number of standards for hardware and software. One such standard is the proposed operating system interface for IMA applications called the ARINC 653.

The advantages of having partitions are:

1. It allows applications with different criticalities to co-exist and run on the same core module without causing any potential damages to other partitions/applications.
2. It allows integrating and reusing applications written on a federated architecture.
3. Partitions provide the flexibility to add enhancement to an application without modifying the schedule or any other applications.
4. The beauty of partitioning is that we can make modifications in one partition and not have to 'regression-test' the others because partition protection is in place.

In traditional federated architecture, each application controlling equipment runs alone (with a RTOS) on a dedicated processor. With the advent of modern processors that have high computing power, IMA supports more than one application on a single processor, thus reducing the cost. Recently the avionics industry is promoting the use of Commercial Off-The-Shelf (COTS) RTOS in aircrafts [6]. ARINC 653 is basically a standard provided to develop operating systems (COTS RTOS) for avionics systems. This standard is slowly getting wider acceptance in the avionics industry.

1.1.2 ARINC 653

The performance of time management studied in this thesis is on an indigenously developed ARINC 653 compliant real-time embedded operating system. ARINC 653 is a standard specification for avionics application software drafted by Airlines Electronic Engineering Committee in 1997 [3]. ARINC stands for Aeronautical Radio INC. ARINC 653 standard is a general purpose APEX (APplication EXecutive) interface for an avionics computer's operating system and the application software. The standard includes interface requirements as well as list of services that allow the application software to control the scheduling, communication and status information of its internal processing elements.

Central to the ARINC 653 philosophy is the concept of partitioning, whereby functions resident on a core module are partitioned with respect to space (memory partitioning) and time (temporal partitioning). A partition is therefore a program unit of the application designed to satisfy these partitioning constraints. The OS is responsible for enforcing partitioning and managing individual partitions. A robustly partitioned system allows partitions with different criticality levels to execute in the same core module, without affecting one another spatially or temporally. The *time partitioning* ensures that activities in one partition do not disturb the timing of events in other partitions. The *space partitioning* ensures that software in one partition cannot change the software or private data of another partition either in memory or in transit.

1.1.3 Partitions

Partitions are scheduled on a fixed, cyclic basis. To assist this cyclic activation, the OS maintains a *major time frame* of fixed duration, which is periodically repeated throughout the module's runtime operation. Partitions are activated by allocating one or more partition windows within this major time frame, each partition window being defined by its offset from the start of the major time frame and expected duration. The order of partition activation is defined at configuration time using configuration tables. This provides deterministic scheduling methodology whereby the partitions are furnished with a predetermined amount of time to access processor resources. A core module may contain several partitions running with different periodicity. The major time frame is defined as a multiple of the least common multiple of all partition periods in the module. Each major cycle contains identical partition scheduling windows. Temporal partitioning therefore ensures each partition uninterrupted access to common resources during their assigned time duration.

Figure 1.1 shows an example of a core module consisting of five applications – System Management, Flight Controls, Flight Management, Input-Output processing and Integrated Vehicle Health Management (IVHM), running for a major frame of 200 milliseconds. Each application is considered as a partition. The duration of each partition and the run-time schedule is pre-determined. The schedule is repeated in every major cycle.

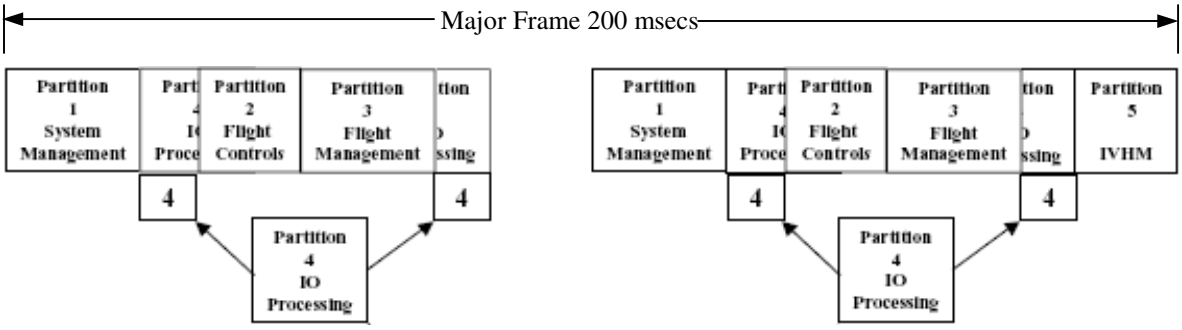


Figure 1.1: Partition window timing requirements

Each core module comprises an operating system running on hardware and is responsible for the scheduling of applications. There can be many such core modules on an aircraft wherein the applications control various equipments of an aircraft. Each such module has a predetermined partition schedule. Configuration of all partitions throughout the whole system is expected to be under the control of system integrator and is maintained with configuration tables. A global schedule has the order of partition activation for each core module. Different constraints such as timing (periodicity), resource and inter-partition communication of each partition are considered by system integrator in the construction of global schedule. This system runs on *time-triggered* bus architecture [4]. "Time-triggered" means that all activities involving the bus, and often those involving components attached to the bus are driven by the passage of time [40]. A time-triggered system controls its own activity and interacts with the environment according to an internal schedule, whereas an event-triggered system is under the control of its environment and must respond to stimuli as they occur. The avionics systems are generally static systems that ease the verification and other stringent processes followed in the industry.

In the current avionics industry, companies like Green Hills Software (INTEGRITY-178B), WindRiver (VxWorks), BAE Systems and LynuxWorks have ARINC 653 compliant real-time operating system. Hence we believe that partitioned system is a field that is growing slowly and has lot of scope for research.

1.2 Time Management

A partition comprises one or more processes (or application threads) that combine dynamically to provide the functions associated with that partition. A process may be periodic or aperiodic. Each process has a priority level. Priority based pre-emptive scheduling is used to schedule these processes within a partition. Thus we have a two-level scheduling model [11] wherein the partitions or applications are statically scheduled based on *cyclic scheduling* method [7] and the processes within each partition are dynamically scheduled based on priority based preemptive scheduling. These processes may use APEX interface or system calls to perform their functions.

Applications running on avionics systems [8] request the service of time management in one way or the other. An application may request for time-out while waiting on a resource, may voluntarily relinquish the CPU for some delay time or have deadline before which it is expected to complete its tasks. These requests must be handled in a deterministic and accurate way with lower overheads. The response time of the system depends on handling these time-out requests along with operating system latency.

The processes can request for time-outs and delays using the system calls. This is how the application's time-out requests get transferred to the operating system. The operating system then calls time management primitives (such as *StartTimer*, *StopTimer* etc.) to handle these requests. The primitives in turn sets and resets the timer provided by the hardware. For example, a process may request for a resource with a specified time-out. If the resource becomes available before the time-out expires, then the process can use that resource. Otherwise, if the time-out expires, then the process not being able to use the resource, may perform an alternative set of actions.

Time management within an operating system uses the hardware timers to service the time-out requests. Most of the latest processors usually provide two kinds of timers, i.e. periodic interrupt timer and one-shot timer. Periodic interrupt timer generates interrupts at a frequency that is programmed in its control register. This frequency is called the tick frequency. One-shot timer also has a control register that is programmed with a count and the timer decrements or increments the count for every timer unit. Once the count reaches a certain value, the timer hardware generates an interrupt. This interrupt is generated once and the control register will be reprogrammed to generate the next timer interrupt.

In a given partition window, the processes may request for several time-out requests when needed. Once requested, the process goes into waiting state until the time-out expires or some other process releases the resource and the time-out request gets cancelled. The action performed on expiry is to make the waiting process available for scheduling by

inserting the process into the ready queue. Then the process gets scheduled depending on its priority.

1.2.1 Time management approaches

1.2.1.1 Tick-based approach

Most real-time systems have traditionally used periodic interrupt timer that generate interrupts at a certain frequency called the tick frequency. This method is known as tick-based approach. In this approach the time-out expiries are handled only when the tick interrupts. In tick-based approach, there are certain efficient algorithms that can be used to handle time-out requests. One such algorithm is the timing wheel algorithm [16] that has O(1) time complexity for all operations performed with the queue. Tick based approach has poor process release delay and therefore poor response time, since the time-out requests are serviced only when interrupt occurs at a given frequency. The accuracy of this approach is defined by the tick frequency. When an interrupt occurs, the current working set of the program (i.e the code and data of the program) in cache may get replaced with the new working set of interrupt service routine (ISR). After the ISR completes, the program continues by bringing back its working set to cache from memory, on a need basis. This operation may consume some amount of time and hence it affects in estimating the *worst-case execution time* (WCET) of the program. As a result, tick-based approach has the problem of frequently disturbing the cache. It also has a substantial interrupt overhead.

1.2.1.2 One-shot timer approach

In one-shot timer approach the interrupt is generated exactly when required by programming a control register with a count. The timer decrements or increments the count in every timer unit. Once the count reaches a certain value, the timer hardware generates an interrupt. This interrupt is generated once and the control register will be reprogrammed to generate the next timer interrupt. This approach provides better accuracy and is very responsive since the timer interrupts can be generated exactly when required. This is based on one-shot timer support provided by the hardware. This approach has the cost of timer reprogramming. Time-out requests can be handled using this approach and there are few algorithms that use this approach. These algorithms perform with the time complexity of O(n) and O(logn). Hence the algorithms that are used in this approach perform poorer than tick-based approach.

1.2.1.3 Firm timer approach

A third approach that can be carefully used in real-time systems is firm timers [18], [19]. Firm timers work similar to one-shot timer except that the interrupt is programmed to generate after an offset amount of time instead at the exact time. This offset is the small delay added to the actual time-out request and is called as overshoot value. The kernel keeps checking for timer expiry at strategic points within kernel, like in system calls, exception handling etc. These checks are called as soft timer checks. If a soft timer check succeeds in detecting an expiry that is supposed to occur at that time, then the time-out request is handled, canceling the interrupt that was supposed to occur after overshoot value. Hence this approach reduces the interrupt overhead. It is similar to soft timer approach [21] except that in soft timer the request can be serviced after an unbounded delay whereas in firm timers this delay is bounded to overshoot value which projects its suitability for a real-time system. If the soft timer check is able to capture an expiry, then the hard timer interrupt is avoided; else the interrupt occurs and handles the expiry. Firm timer approach avoids cache pollution.

1.2.2 Performance parameters

Response time [22] is a critical parameter observed in most real-time systems. In our context, response time is the difference in time at which a process expires to the time at which the first response from that process starts emerging (that is the time at which the process starts executing). Therefore response time is a sum of two components.

Response time = Process release delay + Process scheduling latency

Process release delay (or *release jitter*) is the difference in time at which a process expires to the time at which the process gets inserted into the ready queue (i.e. the process is made available for scheduling). Time management approach and kernel preemptibility control play a significant role in this delay. Process release delay can also be termed as the *accuracy* of timer approach. Process scheduling latency is the difference in time at which the process is inserted into the ready queue to the time at which the process actually starts executing. Time management approach must ensure low process release delay because all the processes in a partitioned system have a deadline before which the process is expected to complete. Poor process release delay may result in a situation wherein a process might miss its deadline leading to catastrophe depending on the criticality of the application.

Hard real-time systems have different requirements than soft real-time systems. *Determinism* and *Predictability* are important characteristics of hard real-time systems. *Constant time algorithms* [23] are usually the preferred algorithms in hard real-time systems

due to the fact that it eases the prediction of worst-case execution time and thereby eases verification of software. Most of the hard real-time system is designed to withstand the worst-case scenario. Thus the worst-case performance of the algorithm is of greater interest than average-case. The hard real-time community ensures that the system remains in stable condition even under worst-case situation and the behavior of the system is reliable.

The design of the kernel also plays an important role in giving better response time to the applications. It should be noted that a high-precision timing facility together with a well-designed preemptible kernel could be the basis for a low latency response system and must be achieved with a low overhead. Unlike in the case of applications running on general-purpose operating system, real-time applications are not throughput-oriented.

1.3 Experiments

The experiments in this thesis are carried out on a truly partitioned system that adheres to strict time partitioning. Chapter 4 explains more on strict time partitioning. The experiments have been carried out on an indigenously developed ARINC 653 compliant real-time operating system called OASYS 653 (Avionics Operating SYStem). Appendix D lists major functionalities supported by OASYS 653 and provides the datasheet given to our customers. Currently OASYS 653 do not support space partitioning due to the fact that the processor (PowerPC MPC 565) on which the OS currently runs does not support Memory Management Unit (MMU). The non-conformance of space partitioning does not affect the experimental results and the analysis, since in a strict time partitioning environment, the timing behavior of individual partitions or applications are independent. Added to this, the scope of our research is primarily time management in partitioned systems.

The results have been collected by running the experiments on a single partition since in our experimental setup and in most partitioned systems, the timing of events in one partition do not affect the timing of events in other partitions. To keep the experimental setup simpler, all the other partitions in our experimental setup are in IDLE mode wherein an idle process executes for the specified partition duration. The partitions are protected in time domain and hence the timing behaviors of the applications running under our partitioned system are independent. Chapter 3 and 4 explains more on these partitioning rules.

Adherence to ARINC 653 standard itself ensures that the real-time operating system developed has all the necessary qualities of an avionics certified operating system. ARINC 653 defines clear-cut expectations for the hard real-time operating system. OASYS 653 have been tested with ARINC 653 defined standard procedures which emphasizes that the platform on which these experiments were carried out has the necessary quality standards. VisionCLICK debugger tools from WindRiver systems are used to collect the experimental

data and the kernel along with the applications are compiled using Diab compiler for MPC 565 processor.

Chapter 6 presents the experimental results primarily with two objectives. The first main section of the chapter compares the performance of the three timer approaches. The partition under consideration has 100% processor utilization that features a fully loaded partition setup. The application running in this partition has a mix of periodic and aperiodic processes. The experimental setup has a rate-monotonic priority distribution and some of these properties are derived from Hartstone benchmark. The experiments collect data on process release delays and the timer approach overheads such as interrupt and soft timer check overhead.

The second main section of Chapter 6 measures the performance of one-shot and tick-based timer algorithms. The time management primitives used in these algorithms are compared and analyzed with respect to space and time complexities. The performance is measured for different values of *n*, which is the total number of time-out requests per partition. The time-out requests expire with a distribution property of Poisson and Bi-modal The experimental setup is designed keeping the timing behavior of a real-time avionics application in mind. In Avionics industry, it is difficult to procure a real-time application due to its confidentiality and cost. Integrating any other real-time application involves making the application ARINC 653 compliant and ensuring that the application stresses the time management module of the OS.

1.4 Thesis Roadmap

This thesis compares the performance of the three time management approaches for a partitioned system. It also presents better one-shot timer algorithms that can be used in real-time systems.

Chapter 2 briefs about the related work and algorithms used in time management. Chapter 3 describes in detail the system model and the partitioning rules. Chapter 4 discusses the timing model and different challenges of time management for partitioned system. Chapter 5 discusses algorithms used under each timer approaches for partitioned system. Chapter 6 measures the performance parameters of timer approaches and algorithms with experiments. Finally, Chapter 7 summarizes the thesis work with conclusions and future work.

Chapter 2

Related Work

There are few interesting literature works in discrete event simulation (DES) [24], [33], [37], that finds its suitability for real-time systems. Most of the algorithms and data structure presented in this chapter are from DES. Heuristic and adaptive algorithms are less favored in hard real-time systems because of its complexity and they are difficult to verify.

The performance of the algorithms is basically measured for three operations. When the time-out request is generated, the request gets inserted into the data structure (Insertion). The primitive used here is *StartTimer*. When the request gets cancelled, the node is deleted from the data structure (Deletion). The primitive used in this case is *StopTimer*. Since almost all the algorithms use double-linked list, deletion always takes O(1) time. On interrupt, the interrupt handler needs to handle the expiry or may do some sort of bookkeeping on the data structure and the primitive used here is *HandleTimeOutExpiry*.

2.1 Tick-based Algorithms

Tick-based approach is based on the notion that all time-outs that expire in a given tick width are handled in the next tick interrupt. When a tick interrupt occurs, the expiries linked for the corresponding tick width are handled in the interrupt handler. Timing wheel [16], and Calendar Queues [17] are the popular algorithms used in this approach.

2.1.1 Timing wheel

In timing wheel there is an array of pointers pointing to a list of expiries. In this algorithm, the data structure into which requests are inserted is an array of lists, with a single overflow list for time-out requests expiring beyond the range of array. Time is divided into cycles; each cycle is N units of time. Hence the array has N entries of pointers to linked list of expiries. In each tick interrupt, the interrupt handler increments the array index and processes

the expiries if the array pointer at current index is non-nil. All time-outs that expire within the current cycle are inserted into the head of the double-linked list pointed by the array. Let the current number of cycles be *S*. If the current time pointer points to element *i*, the current tick is *S* * *N* + *i*. The time-out request *j* is given in ticks relative to the current tick value, say *i*. If *i* + *j* >= *N*, then the time-out tick is beyond the limits of current array and the request is inserted into the overflow list, else the request is inserted into the linked list pointed at index *i* + *j*. The current tick is incremented modulo *N*. When it wraps to 0, the number of cycles is incremented and the overflow list is checked. Any expiries due to occur in the current cycle are removed from the overflow list and inserted into the array of lists.

The overflow list can be kept sorted or unsorted. If the overflow list is sorted then the time complexity for insertion becomes O(n). Otherwise, insertion, deletion and handling a single expiry takes O(1) in timing wheel. Figure 2.1 shows the implementation of timing wheel.

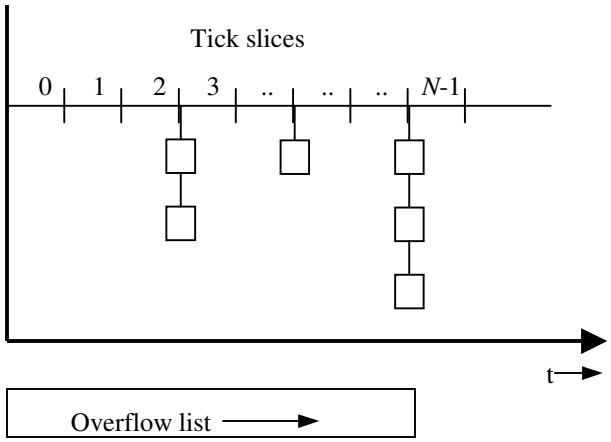


Figure 2.1: Timing wheel

The array can be conceptually thought of as a timing wheel; every time the algorithm steps through *N* locations, it rotates the wheel incrementing the number of cycles. A problem with this implementation is that as time increases within a cycle and the array is traveled down it becomes more likely that time-out requests will be inserted in the overflow list.

If there is a guarantee that all time-out requests are set for duration less than *N*, then a modified timing wheel algorithm takes O(1) latency to perform all the operations. To set a time-out at *j* ticks past the current tick, the algorithm indexes into element *i* + *j* mod *N* and puts the request at the head of a linked list for requests that will expire at a tick = Current Tick + *j* units. This is basically a timing wheel scheme where the wheel turns one array element every timer unit as opposed to rotating every *N* units. In sorting terms, this is similar to *bucket sort*.

The timing wheel algorithm is used in tick-based approach for time management in partitioned system. The partition window is divided into *T* equal sized slice determined by the tick frequency. An array of size *T* is used to point to list of time-outs that expire within the current partition window. More about this algorithm is discussed in chapter 5.

2.1.2 Hierarchical timing wheel

In this scheme there is an array at each level of hierarchy. For instance, one can use 4 arrays; an 100 element array to represent 100 days, 24 element array to represent hours, 60 element array to represent minutes and another 60 element array to represent seconds. This totals to 100 + 24 + 60 + 60 = 244 locations.

For example, let the current time be 11 days, 10 hours, 24 minutes and 30 seconds. Then to set a time-out request of 50 minutes and 45 seconds, the absolute time at which the timer will expire is first calculated. This is 11 days, 11 hours, 15 minutes and 15 seconds. Then the timer is inserted into a list beginning one (11 – 10 hours) element ahead of the current hour pointer in the hour array. The remainder (15 minutes and 15 seconds) is also stored in this location. Ignoring the day array, the setup is shown in figure 2.2.

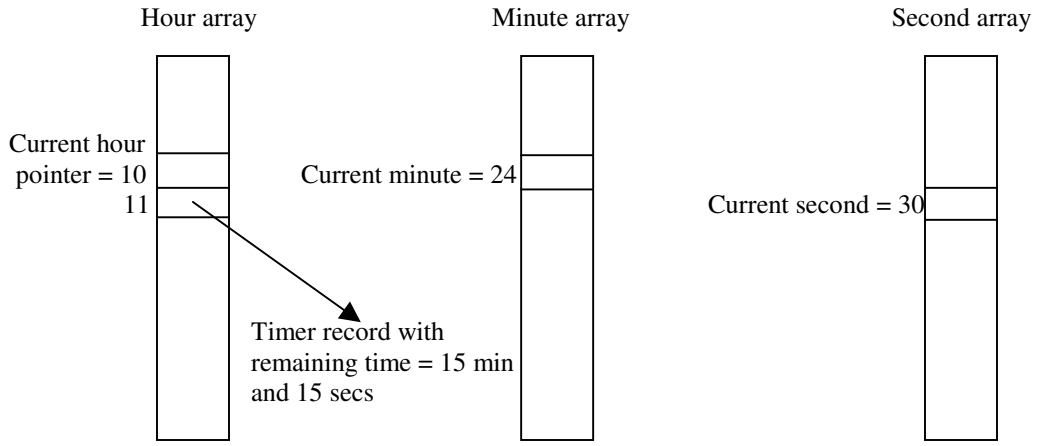


Figure 2.2: Hierarchical set of arrays

Considering that the tick frequency is 1 second, every time the hardware clock ticks, the second's pointer is incremented. If the list pointed to by the element is non-empty, the expiries for the elements in that list are handled. There will be a 60 second timer that is used to update the minute array, a 60 minute timer to update the hour array and a 24 hour timer to update the day array. For instance, every time the 60 second timer expires, the current minute timer is incremented and it handles expiries for minute timer and re-inserts another 60 second

timer. Eventually when the hour timer reaches 11, the list is examined and the remainder of the seconds (15) is inserted into the minute array, 15 elements after the current minute pointer (which is 0). If the minutes remaining were zero, then one can go directly to the second array. Eventually, the minute array will reach the 15^{th} element and the timer is inserted after 15^{th} element from the current value in the second array. Fifteen seconds later, the timer will actually expire.

StartTimer is expected to take O(m) time where *m* is the number of levels in the hierarchy. *HandleTimeOutExpiry* is expected to take O(n * m) where *n* is the number of time-out requests.

A similar idea can be implemented for time management in partitioned system. Given a time-out request we can determine the major cycle where it expires and this forms the first level of hierarchy. In a given major cycle, we can determine the partition window where the request expires which forms the second level of hierarchy. If the time-out expires in the partition's active window, then we can determine the tick slice where the time-out expires, which forms the third level of hierarchy.

2.1.3 Calendar queues

One schedules an event on a desk calendar by simply writing it on the appropriate page, one page for each day [17]. There may be several events scheduled on a particular day, or there may be none. The time at which an event is scheduled is its priority. The insertion operation corresponds to scheduling an event. Scanning the page for today's date and removing the earliest event written on that page removes the earliest event on the calendar. Each page of the calendar is represented by a sorted linked list of the events scheduled for that day. An array containing one pointer for each day of the year is used to find the linked list for a particular day. If the array name is "bucket," then bucket[12] is a pointer to the list of events scheduled for the 12th day of the year. The complete calendar thus consists of an array of 365 pointers and a collection of up to 365 linked lists.

Events can be scheduled for up to one year from the present date by making the calendar circular. The same calendar can be used indefinitely. Notice that there is no need to move pointers or perform any other maintenance on the data structure as one moves from one year to the next. Events can be scheduled more than a year in advance if one writes the date beside each event. Suppose the present date is June 5, 1987. If today's calendar page contains an entry for June 5, 1988, it is clear that this event was scheduled more than a year in advance and that it should not be removed until next year. Before removing an event from today's calendar page, one checks the date written besides it to make certain that it is scheduled for this year. If it is, one removes it and erases it; if not, it is ignored and left for next year or

some other year thereafter. This simple mechanism avoids the need for an overflow list and its associated problems.

The principle of calendar queue is to partition the large list of *n* events into *m* shorter lists and each list is called as a bucket. Bucket is a list with specified range of admission times. Only events that occur in this range are allowed to be scheduled in that bucket. As mentioned in [17], ideally the calendar queue is believed to have O(1) time complexity (for insertion and deletion) for optimal calendar parameters and for uniform distribution of events.

The calendar queue approach is implemented for partitioned systems using one-shot timer, unlike realized in other systems using periodic timer. This implementation requires the hierarchical information of major cycle and partition window where the time-out expires.

2.1.4 Multi-tier linked list (MLIST)

Few calendar queue-based algorithms that have come later such as Dynamic Calendar Queue (DCQ), SNOOPy (Statistically eNhanced with Optimum Operating Parameter) Calendar Queue [28] and Lazy Queue (LQ) tries to improve the cost of resize operation done to reduce the queue length (length of linked list in a bucket) for skewed time-out distributions in order to ensure O(1) time complexity.

MLIST structure [24] works on the principle of three-tier structure to spread out events to handle skewed distributions. This structure is partially derived from a three-tier structure called lazy queue (LQ). Lazy queue includes a near future (NF) linked list or binary heap that is fully sorted, a far future (FF) multi-list that is partially sorted and a very far future (VFF) overflow linked list that is unsorted. The fundamental design of the LQ is to keep only a small portion of the events sorted in NF while letting far future events being unsorted in FF sub-lists and in VFF.

Figure 2.3 shows the structure of MLIST, which is a three-tier structure.

1st Tier (T1): T1 is a sorted linked list. This is the primary tier of MLIST in which the time-out requests that expire in near future are sorted.

2nd Tier (T2): T2 is a partially sorted multi-list structure. T2 stores time-out requests similar to the calendar queue's principle in which time-outs are bucket-sorted. However, within each bucket that houses a linked list, the time-outs are not sorted. MLIST employs a 3rd tier to contain far future requests. Hence T2 is strictly a one-year desk calendar.

3rd Tier (T3): T3 is an unsorted linked list acting as an overflow list to contain far future time-out requests. T3 buffers time-outs that do not affect T1 and T2. This reduces the number of requests in both T1 and T2. In addition, this last tier is the novelty behind the dynamic mechanism of MLIST.

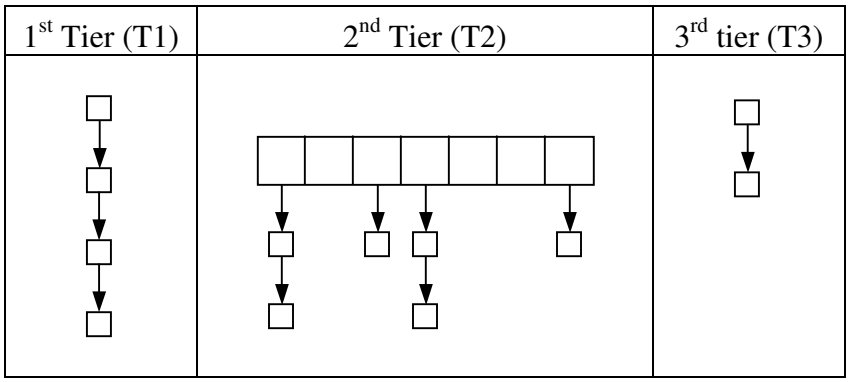


Figure 2.3: MLIST structure

MLIST algorithm uses some set of variables and moves time-outs from T3 to T2 as time progresses and as time-outs in T3 expire within one-year span. The bucket 0 of T2 stores the nearest time-out requests in sorted manner before transferring the same to T1. Eventually time-outs from all buckets in T2 are transferred to T1.

The hierarchical scheme can be combined with the idea of MLIST structure to have a better timer algorithm for partitioned system. More about this is discussed in chapter 5.

2.1.5 UNIX implementation

Hashed timing wheel algorithm with unsorted lists for each bucket is implemented in a version of BSD UNIX callout and timer facilities mentioned in [25]. The time-out request (*c_time*) is expressed in units of clock ticks and is stored in the time-out data structure. A circular array of unsorted lists called *callwheel* is used that contains *callwheelsize* entries. All time-outs scheduled to expire at time *t* appear in the list *callwheel* [*t % callwheelsize*], and their *c_time* members set to *t / callwheelsize*. In each tick interrupt, the appropriate list must be traversed completely and the *c_time* member of each time-out structure is decremented and the expired time-outs are handled and removed. It is claimed in [25] that the average length of the list is a small constant and hence it takes an average-case constant time to process expiries. Insertion is done to the head of the linked list (O(1) time) and hence all timer operations are done in constant time.

2.2 One-shot Timer Algorithms

The data structures for one-shot timers are less efficient than periodic timers. For instance, periodic timers can be implemented using timing wheel which operate in O(1) time, while one-shot timers require priority heaps which require O(logn) time. This difference exists

because periodic timers have a natural bucket width (in time) that is the tick duration between tick interrupts. Timing wheel uses this bucket width and derive their efficiency by providing no ordering to timers within a bucket. One-shot timers have no corresponding bucket width. Unlike tick-based algorithms, a one-shot timer algorithm needs to take care of timer reprogramming to program the timer for the next nearest expiry.

2.2.1 Priority queues

Priority queue is the binary-heap algorithm that can be implemented either as an array or as a tree-based structure. Priority queue helps in maintaining the time-out structure with heap property in a sorted manner. It takes O(logn) for insertion and deletion of time-out request. It takes O(1) for handling single expiry and O(logn) for timer reprogramming i.e. to find the next minimum element or the closest expiry.

Priority queue and other tree-based algorithms attempts to reduce *StartTimer* latency from O(n) to O(logn). It is observed in [16] that this difference is significant for large *n*. Studies on discrete event simulation such as [16], [33] and [37] mentions the popularity of priority queue because; in these systems the tendency to delete a time-out request or an event is very less. Ronnergen and Ayani have analyzed in [33] the performance and complexity of several DES algorithms and suggest that binary heap is a suitable data structure for real-time systems wherein worst-case complexity is a major concern. Priority queue algorithm is suitable if the lowest time-out request gets deleted from *StopTimer* (remove min) always. This cannot be guaranteed in our partitioned system. In such cases priority queue is a poor choice. Added to that if a process gets the requested resource before the time-out expires, then deletion is equally likely to occur and hence O(logn) deletion is a costly effort. Also since the whole system is partitioned and since every partition is required to be handled independently, the possibility on number of time-out requests within a partition is less. Hence we believe *n* is small in partitioned system. Therefore we believe that priority queue performs poorer than linked list based algorithms for partitioned system.

2.2.2 Linux implementation

The APIC timer module of Linux [36] offers high precision one-shot timer to manage time-out requests. These are based on sorted double-linked list. The sorted nature allows issuing timers as fast as possible without the need to search through the complete list. Insertion adds the request to the list by scanning from the end of the linked list and ordering it with the expiry field. The worst-case complexity of insertion is O(n). Timer-reprogramming can be

done in O(1) time. This implementation is used to compare other algorithms in partitioned system.

One-shot timer implementation needs to address the issue of aggregating nearby time-out expiries (simultaneous or closer expiries). Smart Timers [29] proposes that any implementation must have three properties: a) Accurate timing with a settable bound on maximum latency; b) Reduced overhead by aggregating nearby events; c) Reduced overhead by avoiding unnecessary periodic interrupts. The proposed solution for partitioned system discussed in chapters 4 and 5 addresses these issues and others.

2.3 Firm Timers

A firm timer is used to support time-sensitive applications such as multimedia applications in Time Sensitive Linux (TSL). TSL is an extended version of Linux used to measure the performance of firm timers by [19]. Firm timers use one-shot timer and soft timer checks to provide high-precision low overhead time management. The one-shot timer is programmed to expire after an overshoot amount of time.

TSL uses one-shot timers combined with periodic timers to implement efficient firm timers for long time-outs. A firm timer for a long timeout uses a periodic timer to wake up at the last period before the timer expiration and then sets the one-shot APIC timer. Hence, firm timer approach only has active one-shot timers within one tick period. Since the number of such timers, is decreased, the data structure implementation of one-shot timer is more efficient. For periodic timers there are already efficient O(1) algorithms such as calendar queue which is used to store long time-out requests. This approach still has a larger interrupt overhead since periodic interrupts are used. To avoid the overhead of one-shot timer, soft timers are used in TSL.

Firm timers are implemented in partitioned system and experimental results show promising options. The TSL technique of combining periodic timers, one-shot timers and firm timers leads to significant interrupt and time management overhead and hence is not suitable for partitioned system.

Chapter 3

System Model

This chapter describes partition model in detail. It provides an overview of time partitioning and partition-based operating system from the ARINC 653 perspective.

3.1 Background

Every partition may have one or more processes which is also called as tasks or threads in real-time context. The real-time tasks within a partition generally consist of *iterative* (periodic) tasks that must run at some fixed frequency (e.g. 20 times a second) and *sporadic* tasks that run in response to some event (e.g., when the pilot presses a button). Iterative tasks often require tight bounds on *jitter*, meaning that they must sample sensors or deliver outputs to their actuators at very precise instants (e.g., within a millisecond of their deadline), and sporadic tasks often have tight bounds on *latency*, meaning that they must deliver an output within some short interval of the event that triggered them.

There are two basic ways to schedule a real-time system: statically or dynamically. In a static schedule, a list of tasks is executed cyclically at a fixed rate. Tasks that need to be executed at a faster rate are allocated multiple slots in the task schedule. The maximum execution time of each task is calculated, and sufficient time is allocated within the schedule to allow it to run to completion. The schedule is calculated during system development and is not changed at runtime. This is known as cyclic scheduling.

In a dynamic schedule, on the other hand, the choice and timing of which tasks to dispatch is decided at runtime. The usual approach allocates a fixed priority to each task, and the system always runs the highest-priority task that is ready for execution. If a high-priority task becomes ready while a lower-priority task is running, the lower-priority task is interrupted and the high-priority task is allowed to run. The challenge in dynamic scheduling is to allocate priorities to tasks in such a way that overall system behavior is predictable and all deadlines are satisfied. Here, the most popular approach is the *rate monotonic scheduling*

(RMS) of Liu and Layland [14]. Under RMS, priorities are simply allocated on the basis of iteration rate (the highest priorities going to the tasks with the highest rates).

Partition-based system employs static scheduling at the partition level: the kernel guarantees service to every partition for specified durations at a specified frequency (e.g., 20 ms every 100 ms) and the partitions then schedule their tasks within their individual allocations in any way they choose; in particular, partitions may use dynamic scheduling for their own tasks.

A two-level hierarchical scheme for scheduling is proposed in [11] for an open system of multithreaded, real-time applications on a single processor. It allows different applications (or partitions) to be scheduled according to different scheduling algorithms for each partition. A real-time application in the open system can be scheduled according to either a time-driven algorithm or a priority-driven algorithm. If the application uses a priority-driven scheduling algorithm, the algorithm can be preemptive or nonpreemptive, fixed priority or dynamic priority. More importantly, the two level scheme allows the schedulability of each application to be validated in isolation from other applications.

In avionics systems, the number of applications, their resource needs, number of processes in each application, their memory and timing requirements are known well before. These systems do not have online acceptance test.

3.2 Partitioning

The purpose of partitioning is fault containment: a failure in one partition must not propagate to cause failure in another partition. The intent of partitioning is to control the additional hazard that is created when a function shares its processor (or, more generally, a resource) with other functions. The additional hazard is that faults in the design or implementation of one function may affect the operation of other functions that share resources with it. Certification ensures that such faults cannot occur.

Malfunction or unintended function is often more serious than simple loss of function, and the consequences of a propagating fault may well be of these more serious kinds. If the failure of one function could propagate to others, then a low-criticality function might cause a high-criticality function to fail. Partitioning must be used to eliminate fault propagation from low-assurance functions to those of high criticality.

Runaway executions in the kernel, lockups, and untrapped halt instructions could all afflict a processor dedicated to a single function, and so their treatment is more in the domain of system-level design verification or fault tolerance than partitioning. Overruns or runaways within a function, however, are genuinely the concern of partitioning and are usually controlled through timer interrupts managed by the kernel: the kernel sets a timer when it

gives control to a partition; if the partition does not relinquish control voluntarily before its time is up, the timer interrupt will activate the kernel, which then will take control away from the overrunning partition and give it to another partition under the same constraints. Merely taking control away from an overrunning partition does not guarantee that other partitions will be able to proceed, however, for the overrunning partition could be holding some shared device or other resource that is needed by those other partitions. The kernel could break any locks held by the errant partition and forcibly seize the resource, but this may do little good if the resource has been left in an inconsistent state. These considerations leads to a conclusion that devices and other resources cannot be directly shared across partitions. Instead, a management partition must own the resource and must manage it in such a way that behavior by one client partition cannot affect the service received by another.

3.3 Partition based Operating System

3.3.1 System architecture

The software architecture supports partitioning in accordance with the IMA philosophy. Figure 3.1 illustrates the system architecture used in OASYS 653. The underlying architecture of a partition is similar to that of a multitasking application within a general-purpose mainframe computer.

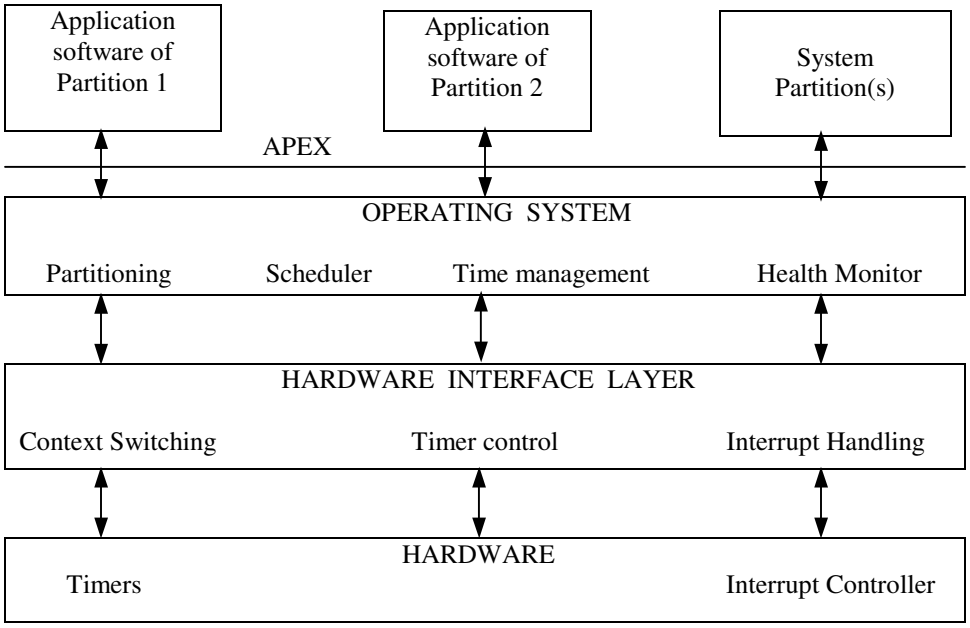


Figure 3.1: System Architecture

The applications or partitions running on the system, call APEX services (system calls) to control various functionalities of the application. OASYS 653 has the following modules implemented.

- Partition management
- Process management
- Time management,
- Health monitoring,
- Interpartition communication
- Intrapartition communication.

The operating system in turn uses hardware interface layer (HIL) services to access the hardware while servicing the application's request. The hardware interface layer makes the operating system independent of the hardware. HIL is a low level layer that is closely dependent on the hardware. It provides services for context switch; interrupt control, timer control and others.

3.3.2 Partition overview

A partition has following attributes:

- Period – defines the activation period of the partition and is used to determine the partition's runtime placement within the module's overall time frame.
- Duration – the amount of processor time given to the partition in every period of the partition
- Criticality level – denotes criticality level of partition.
- Operating mode – denotes the partition mode, such as COLD_START, WARM_START, NORMAL and IDLE.

Figure 3.2 illustrates an example of a typical partition schedule that has a major cycle width of 200ms. There are three partitions in this example. Partition A and B has a periodicity of 100ms. Partition C has a periodicity of 200ms. As soon as a major cycle ends, the next major cycle starts. The window X is called as *slack time* (or free time) during which the system is idle.

There are a total of 8 windows (numbered 1 to 8) in figure 3.2. At run time when the control is transferred to partition **A** in window 1 (during the start of the major cycle), the window is called as the current active partition window. With respect to partition **A**, windows numbered 1, 3, 5 and 7 are active partition windows. The rest of the partition windows are

considered inactive for partition **A**. With respect to partition **B**, windows numbered 2 and 6 are active and rest of the windows are inactive. Note that partition **A** has two partition windows in its period. A partition may have any number of windows in its given period, wherein the whole task that a partition performs for a given period is divided or shared in those windows. Every partition window has a specified duration that is statically determined in the configuration table. A partition cannot execute more than its specified duration otherwise a *partition overrun* exists. The runtime order of each partition as shown in figure 3.2 is known as time partitioning, i.e. the time is partitioned for each applications for a major cycle on a time line. Here the *processor utilization* is not 100% since the system is idle in slack.

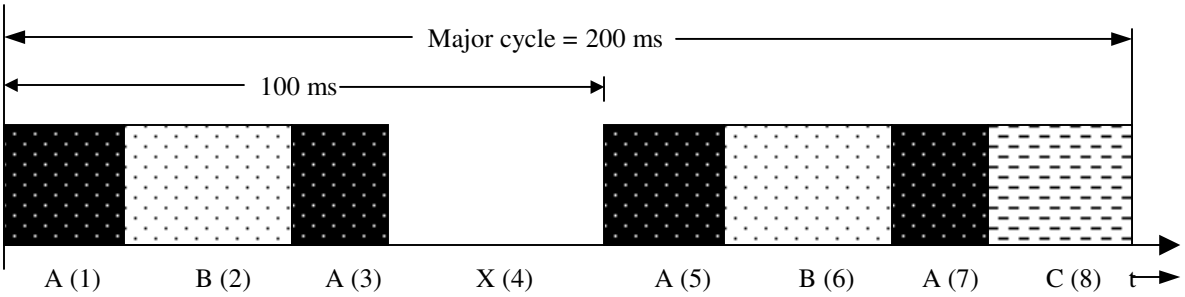


Figure 3.2: Partition schedule

The attributes of partitions are predetermined. Every partition has predetermined areas of memory allocated to it. All the resources such as semaphore, buffers needed by a partition are known well before the system starts and are allocated statically from the memory space reserved for the partition. The number of these resources and the number of processes in every partition is known from the configuration table during the system build time. These resources and processes are created at system initialization. Dynamic memory allocation is not supported in this system. This ensures a higher degree of determinism and the memory management is straightforward. This avoids problems such as memory fragmentation. This also helps to ease certification of the system. It is the responsibility of the operating system to ensure time partitioning and space partitioning.

Processes have the following attributes:

- Period – identifies the period of activation for a periodic process. A distinct and unique value should be specified do designate the process as aperiodic.
- Time capacity – defines the elapsed time within which the process should complete its execution. It indicates the amount of time the process can consume before its deadline expires.

- Current Priority - Defines the priority with which the process may access and receive resources. It is set to base priority at initialization time and is dynamic at runtime.
- Deadline Time - The deadline time is periodically evaluated by the operating system to determine whether the process is satisfactorily completing its processing within the allotted time. For a periodic process, the deadline time is set to its absolute time of release (release point) plus its time capacity. For an aperiodic process, deadline time is set to current system time plus its time capacity.
- Process State - Identifies the current scheduling state of the process. The state of the process could be dormant, ready, running or waiting.

Current priority is the priority used by the OS for dispatching and resource allocation. The deadline time of periodic process is calculated periodically and is set to the next release point plus the time capacity. Release point is the absolute system time at which the process is intended for release. This term is explained more clearly in the next chapter. An aperiodic process can have infinite deadline time. Every partition has a system created idle process that is aperiodic and has the lowest priority. Idle process is executed when there are no processes ready to execute. Idle process has an infinite deadline time. ARINC defines that there could be maximum of 32 partitions in a system. A partition may have a maximum of 128 processes. ARINC defines that the process-scheduling algorithm is priority-based preemptive. If several processes have the same current priority, the OS selects the oldest one. If the end of a partition window interrupts a process within a section of code, it is guaranteed to be the first to execute when the partition is resumed, until and unless there is any other higher priority process released in between. OASYS 653 supports all these requirements.

3.4 APEX Services

There are 56 services that ARINC 653-1 defines and the following section briefs only a few services that are important from time management perspective. All the services are implemented in OASYS 653. The processes that are running as part of application to perform their functionalities, call these services to control the behavior.

- DELAYED_START(*process_id*, *delay_time*) – This service allows a process to be released after a delay time. This service invokes *StartTimer* primitive to request a time-out equal to the *delay_time* value passed. After the time-out expires, the process is released or placed in the ready queue.

- TIMED_WAIT(*delay_time*) – This service suspends execution of the requesting process for a minimum amount of elapsed time (i.e. *delay_time*). The service invokes process scheduler to schedule a high priority process since the current requesting process will have to wait until the time-out expires.

- PERIODIC_WAIT() – This service suspends the execution of the requesting periodic process until the next release point on the processor time line that corresponds to the period of the process. This service also invokes process scheduler. All periodic processes request this service at the end of its task completion for the next periodic activation. This service also invokes *StartTimer* primitive wherein the time-out requested is equal to the periodicity of the process.

- REPLENISH(*budget_time*) – This service updates the deadline of the requesting process with a specified *budget_time* value. It is not allowed to postpone a periodic process deadline past its next release point.

- GET_TIME(*system_time*) – The service returns the value of the system clock from the start of the system. The system clock is the value for a clock common to all processors in the module.

- WAIT_SEMAPHORE(*semaphore_id*, *time_out*) – The service request moves the current process from the running state to the waiting state if the current value of the specified semaphore is zero and if the specified time-out is non-zero. The process will continue to execute if the current value of the specified semaphore is positive and the semaphore current value is decremented.

  ARINC defines a number of similar services that wait on a resource with time-out request. These services in turn call time management primitives to start a timer with the specified time-out. If the resource is not available before the time-out expires, the control is returned to the application that may take an alternative path. Else if the resource becomes available (before the timer expires) by means of other services like SIGNAL_SEMAPHORE, then the timer is stopped or cancelled.

Intrapartition communication has provisions for processes within a partition to communicate with each other. The intrapartition communication mechanisms are buffers, blackboards, semaphores, and events. Buffers and blackboards are provided for general inter-process communication (and synchronization), whereas semaphores and events are provided for inter-process synchronization.

Interpartition communication is a generic expression used in this standard. Its primary definition is communication between two or more partitions executing either on the same core module or on different core modules. It may also mean communication between partitions on a core module and non-ARINC 653 equipment that is external to the core module. All interpartition communication is conducted via messages. The basic mechanism for linking partitions by messages is the channel. A channel defines a logical link between one source and one or more destinations, where the source and the destinations may be one or more partitions. It also specifies the mode of transfer of messages from the source to the destinations together with the characteristics of the messages that are to be sent from that source to those destinations. Partitions have access to channels via defined access points called ports. A channel consists of one or more ports and the associated resources. A port provides the required resources that allow a specific partition to either send or receive messages in a specific channel. A partition is allowed to exchange messages through multiple channels via their respective source and destination ports. The channel describes a route connecting one sending port to one or several receiving ports through intermediate ports.

The *Health Monitor* (HM) is the function of the OS responsible for monitoring and reporting hardware, application and OS software faults and failures. The HM helps to isolate faults and to prevent failures from propagating. This module is not implemented completely in OASYS 653.

3.5 Deterministic Process Scheduler

Traditionally most scheduler work on ready queue that are sorted linked list of PCBs which has indeterministic insertion time of O(n). To over come this problem, we have come up with a deterministic process scheduler that is based on bit-map scheduler [31], [32]. It provides a worst-case insertion, deletion and finding the highest priority process at O(1) time complexity. The ARINC 653 standard supports 64 priority levels for processes within a partition. This enables to have a bit-map scheduler that performs all queue operations in deterministic time and hence the scheduler is called as deterministic process scheduler in OASYS 653. Therefore all operations on the ready queue is performed in O(1) time.

3.6 Critical Sections

Most of the hard real-time operating systems disables interrupts while accessing important kernel data structures such as ready queue, than using semaphores to protect the critical sections. This is because, using semaphores may cause other problems such as priority inversions that requires thorough verification analysis and makes the kernel heavier. In our

case disabling interrupts can create problems as this may delay partition switches, especially if the critical section occurs near partition boundary (and consumes O(n) time). Hence this could violate strict time partitioning. Interrupt service routine may use data structures that are common with the kernel. These data structures are protected inside a critical section, not allowing context switch to happen. The ISR returns if the section is critical and the pending activity of the ISR is later performed at the exit of critical section.

OASYS 653 doesn't use any semaphores to protect its data structures. Basically it ensures that there wouldn't be any context switch while a process is accessing a shared data structure. This also means that a high-priority process will have to wait for a worst-case critical section length, even if it has nothing to do with the data structure protected. The kernel ensures that this critical section length is kept at minimum so as to reduce *blocking time*. In our case primitives such as *StartTimer* and *StopTimer* are protected inside critical sections.

Chapter 4

Timing Model and Analysis

This chapter primarily has two sections. In the timing model section, the chapter describes how time partitioning is implemented in our kernel and justifies on how strict time partitioning is achieved. In the time management analysis section, the chapter analyzes different time management challenges for a partitioned system and how our kernel handles some of them. Finally the chapter discusses various time related parameters that are important to assess a good time management module. Overall, this chapter analyzes the time management concepts for a partitioned system.

4.1 Hardware Timers

The functionalities of hardware timers are essential in understanding the implementation of time partitioning. PowerPC processor MPC 565 with 40 MHz [34] is used in our system. Appendix A gives more details on this processor. This processor has a single-issue core and doesn't have cache and MMU. The presence of cache can have unpredictable delays that can cause problems for verification of avionics applications and other hard real-time applications. PowerPC MPC 565 is one of the widely used processor in avionics industry. In OASYS 653, the timers are set to a resolution frequency of 1µs. The processor supports the following timers:

- **Time Base – TB:** This is a 64-bit free running timer and is divided in to two 32-bit registers for the upper half and the lower half of the register TB. It has a control register that can be used to mask/unmask interrupts. There are two 32-bit reference registers namely TBREF0 and TBREF1, that can be used to generate interrupts when the lower half of the TB register matches with that of TBREF0 or TBREF1. In our case TBREF0 is used to generate interrupt for partition switch and TBREF1 is used to generate interrupts for time-out expiry. This timer is initialized to zero before starting the first partition in the first major cycle. Since it is a 64-bit timer with the count incremented for every 1µs, the

timer can run without overflow for around 500 years. This time is referred as the standard system time in OASYS 653. All activities of the system are timed to this scale. Hence this timer defines the accuracy of the whole system and is a high-precision timer. This timer is used for one-shot and tick-based approach. In order to be in sync with the system time, periodic interrupts are generated using TB in our system

- **Decrementer:** The decrementer is a 32-bit decrementing counter that provides decrementer interrupt. Once the register is initialized with a value, the decrementer starts decrementing the count when the timer is started. Interrupt is generated when the count reaches zero. This timer is used for handling process deadlines in OASYS 653.
- **Software Watchdog Timer (SWT):** This timer can be used to issue system reset or non-maskable interrupt. The software watchdog timer (SWT) prevents system lockout in case the software becomes trapped in loops with no controlled exit. This timer cannot be used with emulator because the emulator does not allow any writes into its control registers. This is to avoid a possible system reset and bus monitor disable by the software, due to which the emulator loses control of the hardware.

The application doesn't have access to any internal hardware timers or registers. Hence all the hardware resources are protected and are handled only by the kernel. The kernel executes in the supervisor mode and the application executes in user mode. The interrupts can be disabled and enabled by modifying a bit value in machine status register. If an interrupt is expected to occur while the interrupts are disabled, then that interrupt will remain as a pending interrupt. This interrupt is serviced immediately once the interrupt is re-enabled. Interrupts are automatically disabled while the kernel is in interrupt service routine (ISR). Hence, nested interrupts are not supported in OASYS 653, which is one of the requirements of a hard real-time system.

4.2 Time Partitioning

There are two major time partitioning rules derived from [1].

1. Under no circumstances a partition can run beyond its allocated duration. In other words partition overrun must not occur. This is termed as strict time partitioning. A partition switch occurs at a predetermined time that switches the context from one partition to the next partition. This partition switch should occur exactly at its intended time. Otherwise the next partition will start with a delay and this may result in a situation wherein the processes in the next partition may miss its deadline.
2. Under no circumstances a partition can perform activities for other partitions. For example, if a time-out requested by partition **A**, expires in partition **B**'s window, then

partition **B** cannot handle that expiry. Otherwise if partition **B** handles partition **A**'s expiry, then it may result in a situation wherein partition **B**'s own processes may miss its deadline due to the delay faced in processing other partition's expiries. This rule is applicable for all other activities.

The partition schedule is driven strictly by the processor's internal clock, so that if an event requires the services of a task in a partition other than the current one, it must wait until the next regularly scheduled activation of the partition concerned. This increases latency, but may not be a problem if partitions are scheduled at kilohertz rates. The currently executing partition should see no temporal impact from the arrival of events destined for other partitions. Even the cost of a kernel activation to latch an interrupt for delivery to a later partition reduces availability of the CPU to the current partition and must be strictly controlled. It is possible to add padding to the time allocated to each partition to allow for the cost of kernel activity used to latch some predicted number of interrupts for other partitions. But this makes temporal correctness of one partition dependent on the accuracy of information provided by others (i.e., the number and rate of their external events) and even originally accurate information may become useless if a fault causes some device to generate interrupts continuously.

Temporal partitioning is predicated on determinism: because it is difficult to bound the behavior of faulty partitions, the availability and performance of each resource is ensured by guaranteeing that no other partition can initiate any activity that will compete with the partition scheduled to access the resource. This means that no CPU or memory cycles may be consumed other than at the behest of the software in the currently scheduled partition. Thus, in particular, there can be no servicing of device interrupts nor cycle-stealing DMA transfers other than those initiated by the current partition.

External events either should not generate interrupts (the relevant partition should poll for the event instead), or it should be possible to defer handling them until the relevant partition is running. Similarly, interrupts due to pending I/O from a device commanded by a previous partition should be masked off. Under static partition scheduling, interrupts from external devices are allowed only when their partition is running.

4.3 Timing Model

The timing model presented here is basically an engineering technique that is implemented in OASYS 653. We assume that most modern processors support high precision programmable timer facility. The timing model is presented for partitioned system using one-shot timer.

Most modern processors support a 64-bit free running timer. At the start of the system, this timer can be initialized with zero and then the timer increments the count for every one unit of time. The application requests for time-outs by passing a 64-bit parameter to the APEX interface. Time-outs are passed as a relative time, i.e. if a process requests to wait for 230 units of time and if the current system time is 30, then the timer is expected to expire at 260 units of time and then awake the process. The relative time can be converted into absolute time by adding it to the current system time. Since the schedule of partition activation is predetermined, the absolute start and stop time of partition window is known. Hence given an absolute time of an expiry, we can determine whether the expiry is inside or outside the current window. Therefore an absolute time of an expiry can fall anywhere on the time scale.

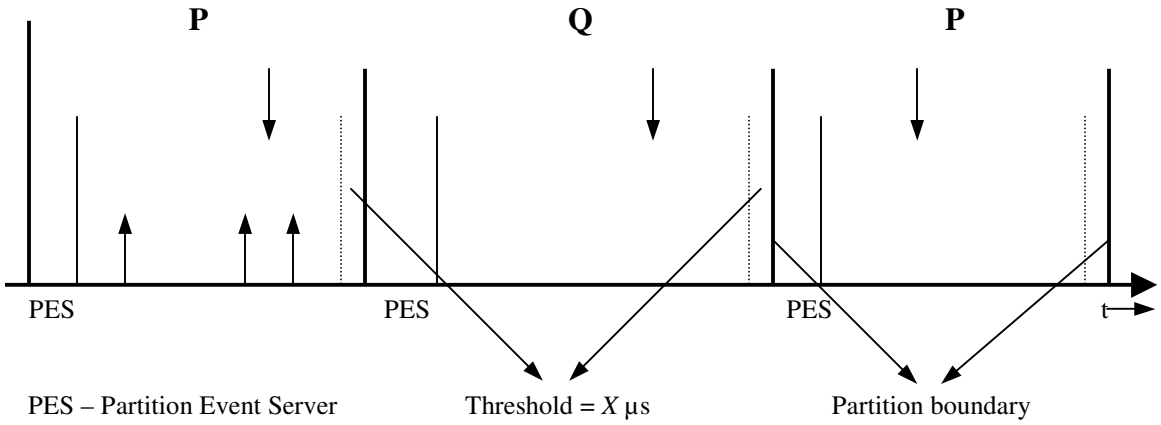


Figure 4.1: Timing model

Figure 4.1 illustrates the timing model of a partitioned system. It highlights a snip of three partition windows of a major cycle. Between invocations of every partition, an interrupt occurs which causes the partition switch. A dedicated one-shot timer is programmed to raise an interrupt at the partition boundary. The partition schedule for a major cycle is statically determined; the start time and duration of each partition window is stored in the configuration table. Upon every partition switch interrupt, the configuration table is indexed and the timer is reprogrammed to generate interrupt at the next partition boundary, by adding the partition duration to the previous partition switch time (with the use of global variables). In our case the partition switch interrupt is generated by using the time base register that provides the standard time for the whole core module. Thus our model ensures that the partition switch interrupt occurs at the exact time. In case of a global schedule, appropriate techniques in bus architectures are used to ensure that the time of a core module matches with that of the global system time (of the entire aircraft).

In figure 4.1, the partition of our interest is **P** and the current window is **P**'s first window. The up arrows indicate the time at which time-out requests are made by processes running in partition **P** and the down arrows indicate the time at which the requests are intended to expire. Three requests are made by partition **P**, the first expire within the partition's current window, second expire in the partition window of **Q** i.e. during partition **P**'s inactive window and the third expire at partition **P**'s next window. As such a process running in a partition can request a time-out that can expire anywhere on the time line. Once the requests are generated, they are placed in a queue. According to the partitioning rules, a partition can handle only its expiries. A partition cannot handle other partition's expiry. Therefore the second expiry during **Q**'s duration is not handled by partition **Q** as the expiry is for partition **P**. This expiry is handled in **P**'s next active window. If the timer expires in the partition's inactive window, it is handled at the next earliest active window of that partition. Thus the process gets released with a delay. It is up to the application to request time-out in such a way that they expire within partition's active window, in order to avoid delay in the process's release. Thus we can clearly notice here that there are two classes of time-out requests. One that expire within a partition's active window and one that expire outside the partition's active window or during partition's inactive window. These expiries are called as inactive expiries.

Partition Event Server – At the beginning of each partition's active window, a *Partition Event Server* (PES) is invoked which handles the expiries that expired before **P**'s current window. It also inserts any requests that are due to expire in the current window of the partition. In figure 4.1, when the control reaches the PES of partition **P**'s second window, it inserts the third time-out request in the timer data structure and programs the timer for the third expiry. The partition event server belongs to kernel and it is the responsible of kernel to invoke the same at the beginning of every partition's window. Partition event server handles all pending expiries and prepares the system for the next partition window. The number of expiries to be handled depends on the expiry distribution of the application. Hence the time complexity of this server depends on the number of such expiries. The worst-case execution time required for PES is not a constant and can be estimated.

The application is deprived of the time consumed by PES in its current active window. The operating system needs to specify the time consumed by PES during the system build time and the application developer must be made aware of this deprivation in order to ensure that none of his processes misses the deadline. The partition switch stores the context of the previous partition (in other words the context of the process that was executing during the time of partition switch interrupt) and transfers the control to the partition event server or the kernel. PES handles the expiries and then restores the context of the next partition (or the process that is due to run in the next partition).

The time-out requested by a process may expire closer to the partition switch boundary. If a timer interrupts closer to the partition switch boundary, then the partition switch is likely to get delayed. This is because the system would be in interrupt mode, inside ISR (servicing the expiry) wherein the interrupts are automatically disabled. This would also disable or postpone partition switch interrupt. This causes problems to ensure strict time partitioning. Therefore in our time partitioning model, timer interrupts are avoided closer to partition boundary. All such time-outs that expire near the vicinity of the partition boundary are considered as inactive expiries. The kernel executes idle process just *X* µs before the partition switch, by generating a special timer interrupt called idle process interrupt (IPI). The system enters into idle process after IPI waiting for the partition switch to occur. This *X* µs is called as the threshold value and should be chosen carefully and is configurable. The application is deprived of executing for this *X* µs.

4.3.1 Time management primitives

There are basically four primitives of time management that OASYS 653 provides to handle time-out requests using one-shot timer.

1. *StartTimer (time, process_id)*
   Inserts the time-out request in the queue for the specified process and programs the timer if needed. This primitive is called inside APEX services.
2. *StopTimer (process_id)*
   Deletes the time-out requested by the process from the timer queue and resets the timer if needed. The time-out is stopped before it expires and reprogrammed for the next expiry, if needed. This can happen if the process waiting (for a certain duration) for a resource gets its resource and hence stops the time-out. This primitive is called inside APEX services.
3. *HandleTimeOutExpiry ()*
   The primitive handles the time-out expiry and enables the process at the head of the queue for scheduling. It sets the timer for the next expiry if needed. This primitive is called from ISR on timer interrupt.
4. *HandleOutWindowExpiries ()*
   All time-out requests that expire before the partition's current window are handled in this primitive. It also inserts the time-out requests to the timer data structure for the current window. This primitive is called only from partition event server (PES).

4.3.2 Strict time partitioning

Strict time partitioning essentially means that the partition switch must occur exactly at the predetermined time. If the interrupts were disabled at the time of partition switch, then there would be an unacceptable delay in switching to the next partition. This problem can also have been handled by using a software watchdog timer (SWT) that can raise a non-maskable interrupt (NMI) at the time of partition switch. In this case the kernel can disable the interrupts because even if interrupts are disabled, NMIs are bound to enforce themselves and cause partition switch. SWT has a different clock source than one used by TB and other timers. Although the SWT can be programmed to have a resolution equal to that of TB, there is still a possibility of having slight variance in generating the partition switch interrupt away from the system time accuracy (which is attained using TB). Hence apart from the fact that emulators do not allow normal software to program SWT, we realize that strict time partitioning can be attained by using normal timer interrupts and time base (TB) is the best option since we can generate partition switch interrupt with accuracy to that of system time.

Ensuring strict time partitioning or having partition switch interrupt at the exact time is a major challenge. In our system, there are three hindrances that can violate strict time partitioning.

1. *Interrupts* – There are two types of interrupts that we need to address here. Timer interrupts and non-timer interrupts. Since application software has no control over the hardware or its registers, it cannot generate interrupts directly by any means. The request must pass through the kernel. The kernel can prohibit timer interrupts at the time of partition switch. Since the kernel knows the partition switch timings, it can easily avoid timer interrupts near partition boundary.

   Similarly any non-timer interrupts that are generated with the kernel visibility can be monitored and prohibited at partition boundaries. But there can be non-timer interrupts such as network interrupts that occur on an arrival of a packet or the interrupt generated while the pilot presses a weapon release button, that pose problems. Partitioning requirements in [1] mentions that the system should not entertain such interrupts that can violate partitioning rules. These interrupts that are intended for a particular partition say **A**, may be generated while partition **B** is executing. Therefore such interrupts must be prohibited at the system build time. The respective partition that is in charge of servicing the interrupt request, polls for the status (packet or button press) at certain frequency. Since these partitions execute at a higher frequency in the order of milliseconds, the delay faced in processing such requests is acceptable. Thus violation of strict time partitioning due to interrupts can be ensured by a careful analysis of interrupt scenarios during the design and system build phase.

2. *Exceptions* – Exceptions are software generated interrupts. In MPC 565 exception have a higher priority than external interrupts. Exceptions may arise due to software errors like divide by zero or when an application in the process of requesting a kernel service cause an exception to switch from user mode to supervisor mode. A simplest way to avoid exceptions near partition boundary is to avoid the application to run near partition boundary. This is achieved, as we know by running idle process just *X* µs before the partition switch. The idle process has no scope of generating exceptions. In this way we can ensure that no application code executes near partition boundary.

3. *Infinite loops inside ISR* – A typical ISR may access data structures and may have a small control loop. If these loops have software errors or bugs, then it may result in a situation where the system enters into infinite loop. This infinite loop occurs inside ISRs wherein the interrupts are already disabled. Thus, this situation can prevent partition switch interrupt to occur and violate strict time partitioning. In hard real-time systems, the whole code undergoes rigorous offline analysis. Therefore, we can safely assume that such problems are removed from the code.

   Similar problem may arise if there is huge number of closer expiries just *X* distance (i.e. threshold value) before partition boundary. In this case, *HandleTimeOutExpiry* primitive attempts to handle all such expiries and may end up running beyond partition boundary. This is avoided by repeatedly checking if the current system time is nearing IPI time and if so, exit.

Figure 4.2 shows the state transition diagram for strict time partitioning which also proves that the approach is deterministic.

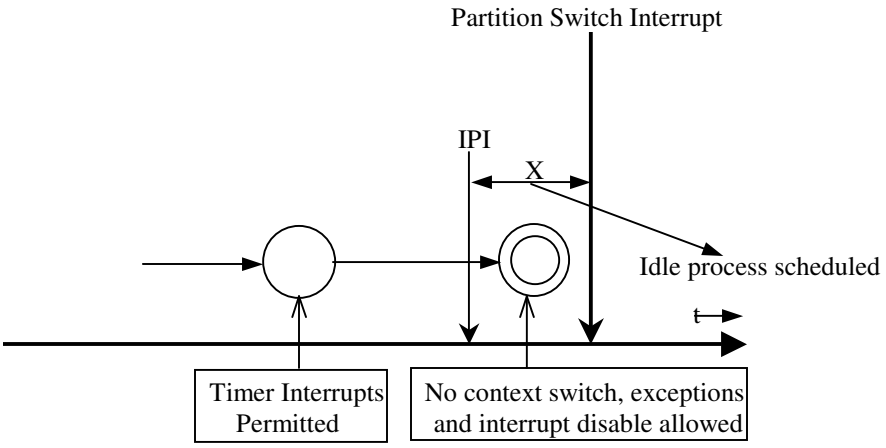


Figure 4.2: Strict Time Partitioning

Therefore by proper and careful engineering techniques, strict time partitioning can be ensured with one-shot timer. From the time management perspective, the kernel needs to take care of interrupts and exceptions. A side effect of ensuring strict time partitioning is that the application is deprived from executing for *X* µs. The partition already looses time to run partition event server. But from the partitioning rules mentioned in [1] we understand that strict time partitioning is more important than the deprivation that the application faces. The OS developer must communicate all these deprivations to the application developer.

4.3.3 Effective available time

From strict time partitioning we understand that the effective available time for the application to execute is much less than the partition window duration (PWD). This is because some amount of time in partition window duration is consumed by partition event server (PES) to handle out window expiries and some amount of time is consumed by kernel (*X* units of time) in order to avoid exceptions at the partition boundary. Therefore the effective available time (EAT) in a partition window for an application to execute is,

$$EAT = PWD - WCET(PES) - X$$

In our implementation, we have found that for small number of time-out requests, WCET(PES) < 200µs and *X* <= 50µs. Now if we consider typical partition window duration of 20ms, then the application has around 98.75% of total partition window duration to execute. For this example, the application is deprived of 1.25% of total partition window duration.

Similarly the time-outs that expire within PWD – *X* part of the partition window duration are considered as the current window expiries by the time management. Any time-outs that expire outside this range are considered as out window expiries. Even the time-out requests that expire on the sharp partition boundary time are considered as out window expiries. Thus our time management deterministically ensures that no time-out requests remains unprocessed.

Figure 4.3 shows the effective available time for an application to execute as well as the time window for current window expiries. Let us assume that partition **A** is of our interest. The partition windows where partition **A** is active in a major cycle is called as the *active partition window*. The partition windows where partition **A** is inactive is called as *inactive partition windows* for partition **A**. When the control transfers to a partition window, we refer to it as the *Current Partition Window* (CPW). Within current partition window duration, only some part of the window is allowed to handle expiries and wherein the interrupts can occur.

This window is called as the *current window* and the time-outs that expire in this window are called as *current window expiries*. The time region outside the current window is called as *out window* and the time-outs that expire in out window are called as *out window expiries*.

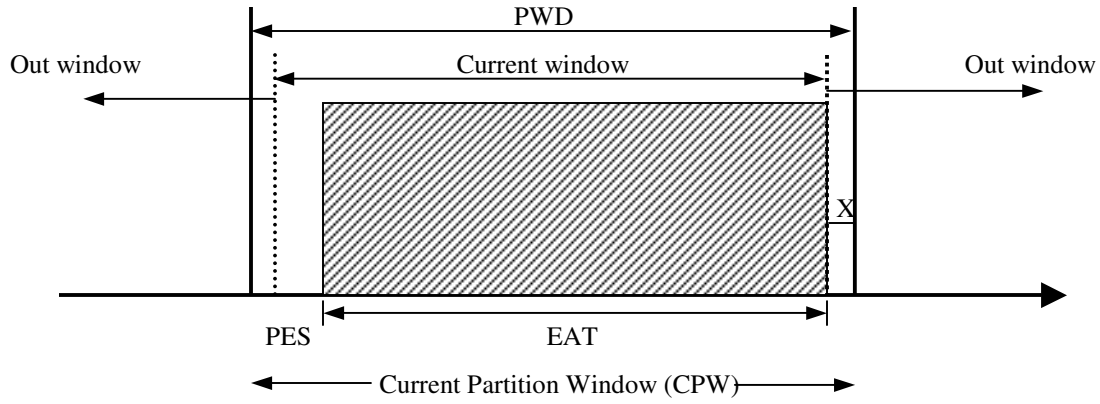


Figure 4.3: Effective available time

4.4 Deadline Management

A process within a partition may or may not request a time-out, but all processes have a deadline before which they are expected to complete their processing. A timer is also used to handle these deadlines. If the timer set for a deadline expires, then the process misses its deadline and the situation is handled appropriately. ARINC 653 defines that the application developer needs to create an error handler process, which handles the deadline miss by either stopping the process that missed the deadline or by re-initializing the process or sometimes it could be even ignoring the deadline miss, depending on the criticality of the application. Error handler process executes with highest priority and is non-preemptible. Each partition has its own error handler process.

Under most circumstances, it would be very rare that a process would miss its deadline after thorough verification analysis. But when a deadline miss occurs, it could result in a serious catastrophe depending on the criticality of the partition. Hence even though a deadline miss is a rare case, it must be handled immediately with low overhead and deterministically. A process that has missed the deadline must be stopped at the earliest; otherwise there can be a possibility that more processes tend to miss their deadline.

Considering the common case that a deadline miss is very rare, a simpler and a lighter approach can be adopted for deadline management, that favors O(1) time for insertion and deletion of deadline requests. Deadline management in OASYS 653 is implemented using

sorted double linked list, which takes O(n) time for insertion and O(1) time for deletion and handling deadline expiry.

4.5 Time Management Analysis

This section analyses time management concepts with the practical implementation perspective for partitioned systems, in our kernel. Since the partitioned system under study is used for avionics, we first try to understand the behavior of a typical avionics application and later analyze different time management concepts. The analysis presented here is using one-shot timer implementation.

4.5.1 Avionics applications

IMA and its partitioning principles have evolved recently. Therefore we believe any details of avionics applications on a partitioned system are yet to be known. Since IMA encourages the re-use of applications from a federated architecture, we present here a case study from [15] and analyze its suitability for partitioned system.

The paper by Locke [15] presents Generic Avionics Platform (GAP) timing requirements of an avionics system consisting of eight applications and its processes with their periodicity and execution time in milliseconds. GAP is a model of an avionics mission computer system similar to certain existing US Navy aircraft. Table 4.1 presents GAP timing requirements.

Major systems modeled in GAP are:

- Navigation computes aircraft position, altitude etc.
- Radar control returns target position.
- Radar warning receiver provides threat position.
- Weapon control, when activated, updates weapon ballistics.
- Display updates the screen information.
- Tracking updates target information
- Built-in-Test determines equipment status.
- Data bus performs communication between mission control computer and devices external to MCC.

The total processor utilization of the system is close to 85%. The weapon control system is activated by an aperiodic event, like a detection of a pilot button depression requesting weapon release. For the weapon protocol subsystem, the time specified is a response time

requirement, for the weapon release and weapon aim subsystems the time specified is a periodic rate to be used until the subsystem completes its required function.

System	Subsystem	Periodicity (ms)	WCET (ms)
Display	Status update Keyset Hook update Graphic display Stores update	200 200 80 80 200	3 1 2 9 1
Radar warning	Contact Management	25	5
Radar	Target update Tracking filter	50 25	5 2
Navigation	Navigation update Steering commands Navigation status	59 200 1000	8 3 1
Tracking	Target update	100	5
Weapon	Weapon protocol Weapon release Weapon aim	200 (Aperiodic) 200 (Aperiodic) 50 (Aperiodic)	1 3 3
Built-in-Test	Equ. Status update	1000	1
Data bus	Poll bus devices	40	1

Table 4.1: GAP timing requirements

Iain Bate in [12] presents a task set consisting of 71 periodic tasks in a major time frame of 200,000 units for Rolls Royce Electronic Engine Control System. All the tasks run with a certain periodicity and have a deadline that is equal to its periodicity.

From the above cases, one can infer that avionics applications have periodic processes at large and few aperiodic processes that are required to execute on certain events. However, different applications may have different needs. Considering that major avionics systems today encourage *rate-monotonic scheduling* [1], high frequency periodic processes have higher priority than low frequency periodic processes. Normally a periodic process has its relative deadline set equal to its periodicity but a few highly critical application may set the relative deadline less than the process's periodicity. Generally we may assume that aperiodic process gets started by a periodic process that is polling for an event. These processes handle the event and then stop themselves. Some aperiodic process may engage in doing house keeping activities and these processes may run for the complete lifetime of the system. Such aperiodic processes have an infinite deadline limit and run with a low priority.

A good time management scheme must try to exploit the timing behavior of an application, but having said that we understand that in a partitioned system, there can be

different applications of different criticalities with different timing behavior running in each partition. All these applications run on a single kernel and hence the kernel cannot be tailored to have a time management algorithm that suits a specific application. In the following sections we analyze the timing behavior of a common case hard real-time application with a one-shot timer, assuming that there is a higher probability that a time-out will expire.

4.5.2 Typical time-out requests

Applications may have different timing behavior and by studying the timing behavior of these applications, one can generalize the property of timing behavior. In this section, we try to analyze how in general an application generates time-out requests within its partition window.

In a hard real-time scenario, we believe that an application developer must be certain on what relative time-out value he needs to pass to a service. An application may have processes that communicate with each other and hence are dependent on each other. Such processes are called as co-operating processes. The relative time-out is a value calculated based on factors such as, what other co-operating processes needs to run before the time-out expires, periodicity of the process and partition concerned, deadline of the process etc. A process will typically request a time-out to allow other processes to run such that, the duration of wait is sufficient for these dependent processes to run in between and perform their part of the work. In other words, by the time the time-out expires for a particular process, that process is certain that all the other dependent processes would have completed its task, so that the current process can continue smoothly.

Conscious timing: If an application has sensitive deadline requirements, then the application developer, while coding a process makes a conscious effort to give a suitable time-out value in a service considering various factors. This is the reason why an application developer must also be sure if the time-out will expire within the partition window or outside and that the process will not miss its deadline. He may also avoid inactive expiries, since the process gets released with delay. In the final setup of the application, each time-out value passed to a service is based on a pre-conceived notion of the application's or its processes timing behavior. A rigorous verification analysis will also be done to ensure that no process will miss its deadline.

Non-conscious timing: When partitions are scheduled in faster rates i.e. in the range of milliseconds, then process release delay may not be an important factor. For some applications, it might be permissible if processes have release delay in the order of

milliseconds, since the major cycle length itself is in milliseconds. A process expired in an inactive window may soon be released in the next active window and this delay is acceptable, especially for low criticality applications. In such cases, the application developer needn't be aware of the partition window timing constraints. He only needs to specify the time-out value in a permissible range, such that the process never misses its deadline.

The above observation is applicable when the time-out is requested by a service that has a higher probability of expiring. The situation can be slightly different in case a time-out is requested by a service that has lower probability of expiring. For example, let us consider that an application waits on a semaphore using the WAIT_SEMAPHORE service by passing a suitable time-out value. Most hard real-time applications are ensured to have enough resources for the smooth running of its processes. This would mean that the requesting process has a higher probability of acquiring the resource before the timer expires and the timer gets cancelled. If the timer expires, then it would mean that the process has failed to acquire the requested resource.

Observation: Intuitively we may find that in a partition window, as we move forward on the time line, there is a higher tendency that a time-out requested by a process will expire farther down the time line. In other words, if we assume that the relative time-out requested by each process is a constant, then we can be certain that each time-out request will be inserted after the last element in the linked list, since the linked list is sorted based on the expiry time. Figure 4.4 shows a typical scenario for this in a partition window.

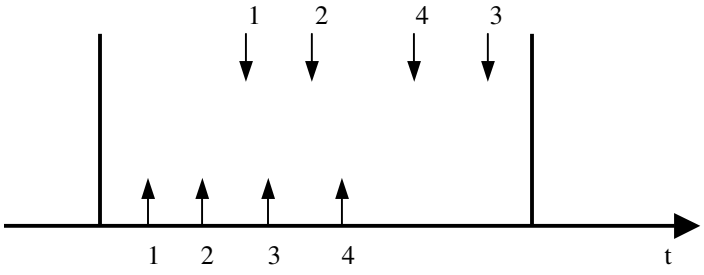


Figure 4.4: Typical time-out requests

In figure 4.4, request **1**, **2** and **3** gets inserted at the end of the sorted linked list. Request **4** gets inserted at the head of the linked list because by the time request **4** is generated, the linked list has only request **3**; **1** and **2** would have expired. Here the system has a higher probability that new requests will be inserted at the end of the queue. Therefore we can

exploit this observation to reduce the insertion time by scanning the linked list from the end (instead of scanning from the beginning) to insert a new time-out request.

Proof: When a process **A** requests a time-out, then in case of a co-operating process environment, the requested relative time-out value must ensure that other co-operating process gets sufficient time to execute before the timer expires, otherwise the co-operating process may not be able to release a resource (if any). This would in turn facilitate the process **A** to continue smoothly later. By contradiction, if the requested time-out value is shorter, then it would give little opportunity for other dependent processes to run and hence the timer would indeed fire. Therefore in order to give more time for the co-operating process to execute, it is essential that the time-out value should be large or in other words expire farther down the time line.

4.5.3 Periodic process release

This section analyzes the timing behavior of periodic process. Figure 4.5 shows a periodic process that is released at the start of the partition window. The partition executes with a periodicity of 100ms in a major cycle length of 200ms. Therefore the periodic process has a periodicity that is a multiple of the periodicity of partition, i.e. 100ms. The periodic process has a relative deadline of 100ms, which is equal to its periodicity.

In figure 4.5, the periodic process gets released at **R**. It executes for some duration and it may request for a time-out at **TO** (hence dotted arrow). If it requests for a time-out, then the process is made to wait till the time-out expires at **E**. Now assuming that this process has highest priority, it executes for a while, till it waits for its next periodic release by issuing PERIODIC_WAIT service at **PW**. The next periodic release is at **R** in the next partition window, which is also the absolute deadline **D** for the current instance of the periodic process.

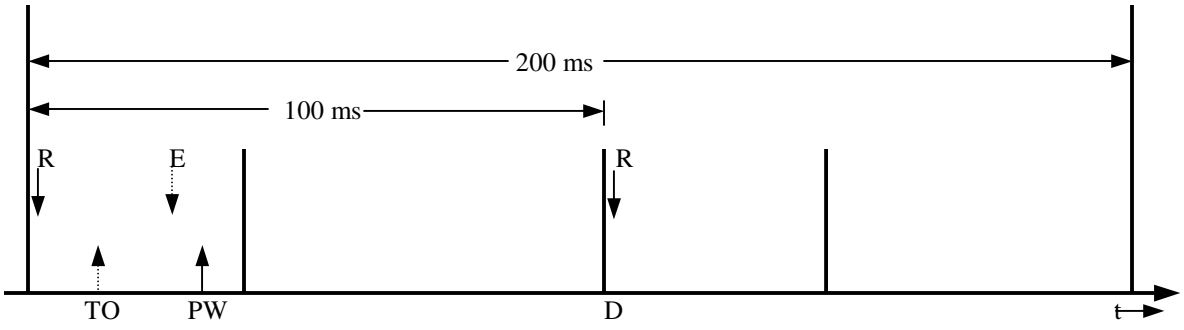


Figure 4.5: Release of a periodic process

Observing this example closely reveals few interesting facts.

1. Once the periodic process is released at **R**, it must execute and issue PERIODIC_WAIT before the partition window ends, because otherwise the process may miss its deadline since it has no opportunity (or partition window) to execute before its deadline expires.

2. The process may request for time-outs any number of times before issuing a periodic wait. All these time-out requests must expire within the current window in case of figure 4.5, because otherwise the process may miss its deadline. Whatever be the case, a process cannot request a time-out that expires after its deadline expires. For the above example, even though the actual absolute deadline for the process under consideration is at **D**, but practically the deadline is at the end of current partition window. Therefore the relative time-out request must not expire beyond this practical deadline since the partition has no windows to process the expiry before **D**. From this we can infer that in a partitioned system, the effective or practical deadline of a process can be much lesser than its absolute deadline. This is because the effective total time that a process gets to execute is much lesser than what it has in a non-partitioned system, due to partition window constraints.

   If d is the relative deadline of a process and *IPD* (for inactive partition duration) is the total duration allocated for other partitions before d expires, then the process is deprived of *IPD* amount of time for execution before its actual deadline expires. Hence the effective deadline of the process is reduced to $d - IPD$.

4.5.4 Harmonic periodic releases

This section analyzes the harmonic periodic releases of periodic processes for a given partition. This analyzes helps in understanding the release behavior of periodic processes and their expiry distribution for a given partition. Table 4.2 consists some set of periodic processes whose periodicity is harmonic in nature and has a rate-monotonic setup. The rate-monotonic setup can be considered as the common case existing in most hard real-time systems. In case of harmonic periodic release, the process's periodicity is an integer multiple of some fundamental periodicity.

For the same partition setup shown in figure 4.5, figure 4.6 shows the harmonic periodic releases of the partition for the example considered in table 4.2, for up to four major cycles. Let us assume that all periodic processes have relative deadline equal to their periodicity.

Types	Processes	Periodicity
A	P1, P2, P3	100 ms
B	P4, P5, P6, P7, P8	200 ms
C	P9, P10	400 ms
D	P11	800 ms

Table 4.2: Harmonic periodic setup

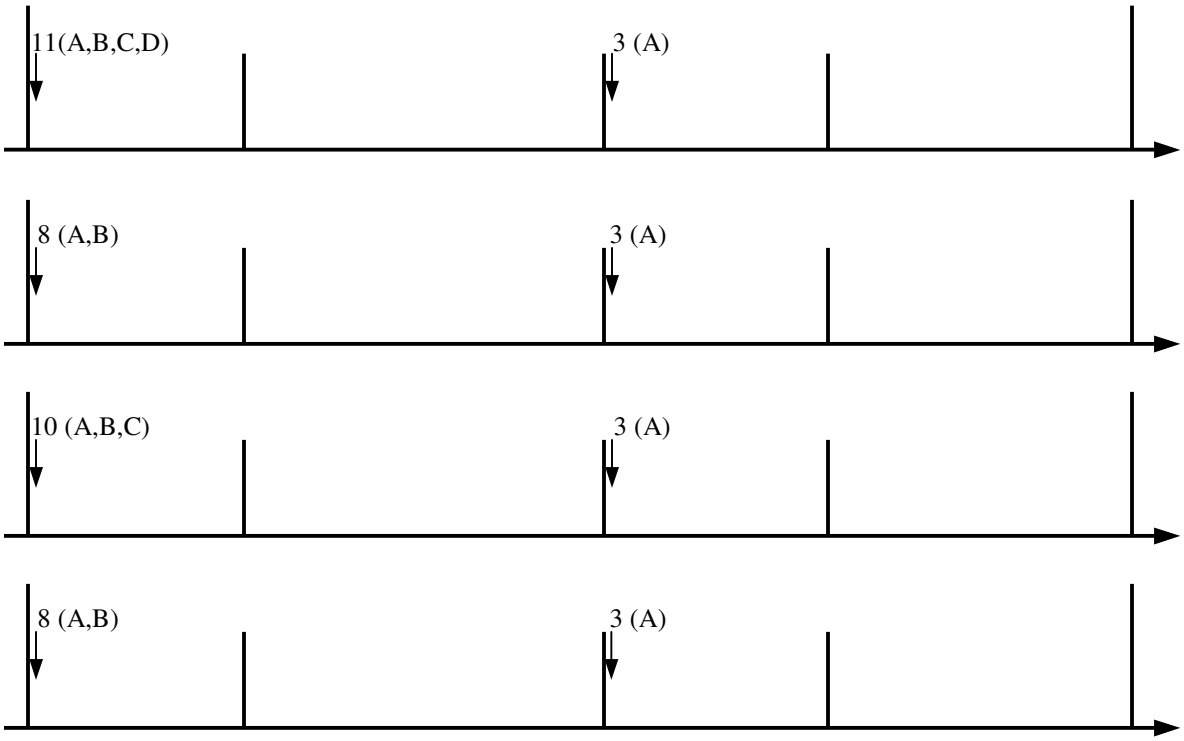


Figure 4.6: Harmonic periodic releases

All the processes are released in the first window of the first major cycle. The numbers in figure 4.6 indicate the total number of processes released at that instant. At the time of release, high priority process from type **A** faces a delay due to lower priority processes from other types. At this instant of release, all the processes are inserted into the ready queue, and a high priority process has to wait until all the processes are inserted. This instant where there are simultaneous release of large number of processes is called as *critical instant* because at this instant a process faces maximum response time. From figure 4.6, we understand that there is a specific release pattern that keeps repeating and hence we may have a predictable environment for the lifetime of the system.

High priority process such as the one in type **A** of the above example can request for time-out that expire in the current partition window. Processes in type **B**, can request time-out that

expire in the current major cycle and processes in type **C**, can request time-out that expire anywhere in the next two major cycles. Processes in type **D** can request time-out that expire across four major cycles. In a rate-monotonic setup, a high priority process executing frequently is likely to have a conscious timing effort with minimum release delay, whereas a low priority process can afford to have a non-conscious timing effort.

A very subjective analysis done for the above example suggests that, at the start of every partition window, there is a high probability of expiry processing due to process releases. At any given instant of time, the probability that a time-out expires in the far future reduces. Hence, for the above example, any time management scheme that manages near future expiries (falls within a major cycle width) efficiently is better, since far future expiries doesn't come under the common case.

The kernel can handle time-out expiries in slack window, no matter for any partition. This reduces the burden of handling them in partition event server. ARINC 653 defines two services for interpartition communication that allow the application to pass time-out requests and have similar behavior to that of WAIT_SEMAPHORE service. These services may affect the expiry distribution of the partition with which it communicates. The timing of events in one partition may affect the timing of events in other partition only if they are involved in interpartition communication. Otherwise the expiry distribution of a partition is independent of other partitions in which case analysis of each partition can be done in isolation due to the fact that partition protection is in place. The expiry distribution affects the timer queue length and therefore the insertion time of *StartTimer*. It also helps in choosing a better algorithm for managing time-out requests.

In a partitioned system, the expiries of the whole system are partitioned and therefore '*divide and rule*' property is very much evident. A good time management scheme must look at several factors of the system and adopt a suitable approach and algorithm. Therefore it is important to understand closely the timing behavior of the system and design a better time management module.

4.5.5 Service Latency

When the relative time-out request is passed by the application through APEX service, the kernel will have to do some amount of processing (error checking etc.) as part of the service before actually starting the timer by means of *StartTimer* primitive. The time taken to do the processing will depend on the APEX service, as each service requires different amount of processing. Service latency is the difference between the time at which the actual call to the APEX service is made by the application and the time at which the absolute time-out is calculated after which the timer gets started inside the *StartTimer* primitive.

Figure 4.7 shows the time where the time-out request is made to the time where process is released. The example considered in this figure has a service latency of 6µs. The APEX service call is made at a system time of 100µs. The service call passes 300µs as the relative time-out parameter. With a service latency of 6µs, the *StartTimer* primitive is called at 106µs (to start the timer) and therefore the absolute time-out set = 300 + 106 = 406µs. The process is now put into the wait queue and is waiting for the timer to expire, for a duration of 300µs. Timer interrupt is generated at absolute system time of 406µs. After this the ISR calls the *HandleTimeOutExpiry* primitive to handle the expiry and inserts the process into the ready queue. The delay faced due to this is called the *process release delay*.

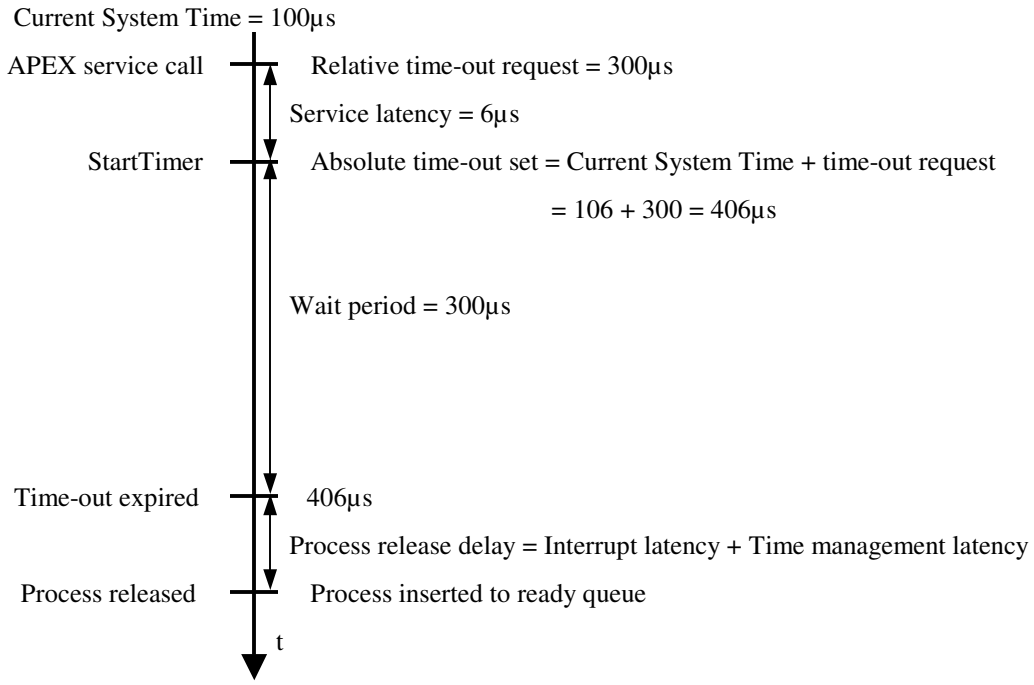


Figure 4.7: Time-out request to process release

The service latency is an inherent delay faced by the service, which the application developer should be aware of. Note that the relative time-out passed into the service is passed directly without modification to the *StartTimer* primitive. We believe that service latency is not loss of accuracy because this latency is part of the service request that is well documented in all the services of ARINC 653 and is accepted.

4.5.6 A Closer look at expiry

This section explains the latencies involved when a timer expires. Figure 4.8 depicts the picture of what happens when a time-out expires. It shows the details about expiry processing

and the inherent delay faced for the process release. As soon as timer interrupts, ISR prologue of the interrupt handler is invoked which stores the context of the current process executing into the respective PCB (Process Control Block). The control is then transferred to Interrupt Service Routine (ISR). ISR calls *HandleTimeOutExpiry* primitive to process the expiry. The primitive finds the process that has expired from the head of the timer queue linked list and inserts the same into the ready queue. Then the primitive reprogrammes the one-shot timer for the next nearest expiry if any. Before the ISR exits, a scheduling decision is made to choose the next highest priority process that is due to run. Finally when the ISR exits, ISR epilogue is invoked which restores the context of the current high priority process that is supposed to execute next. Thus if the priority of the process **H** just expired is higher than the priority of the process **B** that was executing before timer interrupted, then process **H** will execute when ISR exits, else the previous process **B** will continue to run.

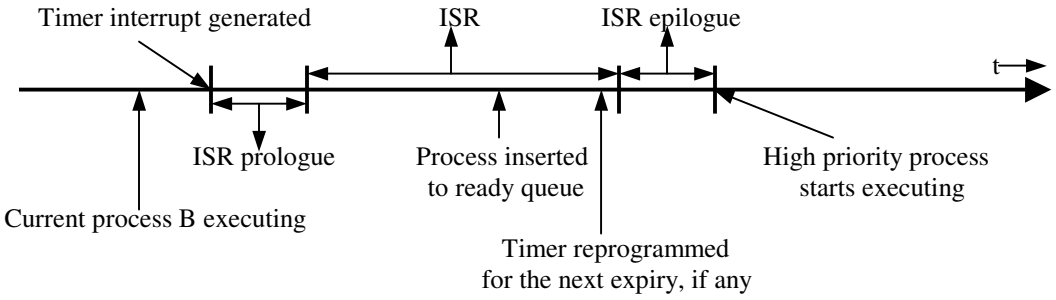


Figure 4.8: Expiry processing

Therefore the ISR performs two activities, one is the expiry processing and the other is the scheduler decision on the next process to be scheduled. The expiry processing belongs to time management module and therefore is considered as the *time management latency*. The decision on which process should be scheduled next is made by the scheduler component in the ISR and hence is termed as the *scheduler latency*. The time consumed by ISR prologue and ISR epilogue forms the third component of latency called the *interrupt latency*. All the three latencies are inherent part of handling every expiry. The scheduler latency remains almost fixed and takes O(1) time due to the deterministic scheduler implemented. Process release delay comprises the interrupt latency, time management latency and the time taken to insert the process to the ready queue which is O(1) in our case. Ignoring the O(1) insertion time to ready queue,

Process release delay = Interrupt latency + Time management latency

4.5.7 Expiry inside a critical zone

A critical zone is one where a context switch or access to critical data structures by any code other than the one executing is prohibited. In our implementation, the critical sections and the error handler process are considered as the critical zone. Timer interrupt is allowed to occur inside a critical zone but it returns immediately without any action noting that the zone is critical. Once the critical zone exits, the pending action is performed. As we understand here the process faces a release delay up to a WCET of critical zone. This is an inevitable situation and can be made better by reducing the length of the critical zone. An error handler process is rarely invoked unless there is a deadline miss or if an exception occurs. There can be at the most one expiry inside any critical zone for the simple reason that any further expiries requires reprogramming of the timer which is not done until the critical zone exits. Here the process release delay is the same as earlier but with an additional delay of the worst-case critical zone length.

$$Process\ release\ delay^{CZ} = Process\ release\ delay + WCET(critical\ zone)$$

For a hard real-time system, it is important to predict the critical section length. This can be done only if the critical section takes O(1) time to execute. In our kernel implementation, time management primitives such as *StartTimer* and *StopTimer* are called inside the critical section, which do not have a constant time for one-shot timers. Insertion and deletion from any internal kernel queues occur inside the critical section. ARINC 653 has designed the specification to favor constant time operations for internal kernel queues. Therefore it becomes more important to have constant time algorithms for timer queues to deterministically predict the critical section length.

4.5.8 Closer expiries

This section explains on how closer expiries are handled. Closer expiries are those expiries that occur within a small delta amount of time. As we saw in section 4.5.4, closer expiries occur at critical instant. Handling closer expiries requires special attention especially in case of one-shot timer. The problem of handling closer expiries can be understood from figure 4.9.

An interrupt is generated for an expiry and the ISR calls *HandleTimeOutExpiry* primitive to handle the expiry. To be on safer side, a check is done inside the primitive to ensure that the interrupt generated is for the expiry at the head of linked list (timer queue). It then handles the expiry by deleting the time-out request from the head of linked list, inserts the expired process into the ready queue and then reprogrammes the one-shot timer for the next

nearest expiry. In figure 4.9, we can clearly observe that by the time the primitive reprogrammes the one-shot timer for the next nearest expiry (i.e. for **1**), the time at which that expiry is supposed to occur has already passed. Thus the one-shot timer programmed never fires and all subsequent expiries will remain undetected. This has happened because the nearest expiry lies before the time timer is reprogrammed.

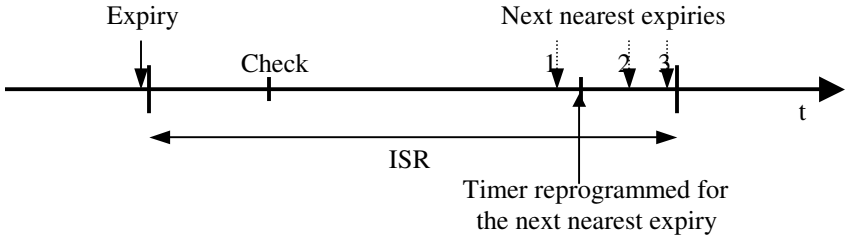


Figure 4.9: Closer expiries

The problem can be solved if *HandleTimeOutExpiry* primitive loops to handle closer expiries by checking if the time-out at the head of linked list is less than the current system time plus a constant time. This constant time is the time taken to execute the fall through path of the loop till timer reprogramming, which can be estimated and is a fixed value for a processor. The primitive tries to handle time-outs that expire before the timer is reprogrammed. Note that as the primitive loops, we are moving forward in time and hence the check dynamically handles all such closer expiries. Therefore, closer expiries are aggregated by dynamically checking for the expiries inside the primitive or ISR. The loop also checks dynamically if the primitive is moving closer to partition boundary (IPI), in which case, the loop exits immediately. This is to ensure strict time partitioning. In such a case unprocessed expiries of closer expiries are handled in the next partition window.

Algorithm: HandleTimeOutExpiry

Input: Linked list of expiries with Head pointing to the head of linked list, Absolute End time of current window

Output: The expiries are handled by inserting the process to ready queue

```
While ((Head is non-nil) AND
        (Current_System_Time < Absolute End time of current window) AND
        (Time-out at the Head of linked list < Current_System_Time + Constant))
          Handle the expiry by inserting the process to ready queue;
          Head = Head->next;
Re-program the timer to the time-out at the Head of linked list, if any;
```

In case of expiry **2** of figure 4.9, the primitive programs the one-shot timer for the expiry, since it occurs after timer reprogramming. A reference register match will force an interrupt. But since we are in interrupt mode (and interrupts are disabled while in ISR), expiry **2** remains as a pending interrupt until ISR exits. As soon as ISR exits, interrupt is again generated to handle expiry **2** and the system enters back into interrupt mode.

An inherent problem of handling closer expiries in this way is that if a high priority process is inserted into the ready queue before others (as part of closer expiries), then it may have to wait for all lower priority processes to get inserted into the ready queue. Thus a high priority process **H** faces a delay to start its execution due to other lower priority processes. If a strict deadline is set to **H**, then it may miss its deadline.

If the process that was executing before closer expiries were processed, say **B** is itself the high priority process even among the processes released in closer expiries, then process **B** also faces a significant delay due to the fact that process **B** is deprived of its execution due to the amount of time consumed by ISR to handle closer expiries. This may affect process **B** and it may miss its sensitive deadline. We believe that the deadlines of the processes are set in such a way that they can still continue without missing their deadline in spite of time consumed by closer expiries.

4.6 Parameters

It is understood from the timing analysis that a good time management alone cannot be the bottleneck for an optimum performance of the system. The kernel also plays a major role in providing optimum system performance such as in preemptibility control. Along with performance, verification also plays a major role in hard real-time systems. Factors such as determinism and predictability are considered important in hard real-time systems.

4.6.1 Determinism

It is well known that any algorithm used by hard real-time systems must be easy to verify. An OS developer needs to specify some important parameters such as the worst-case execution time (WCET) of an APEX service, the interrupt latency, WCET of PES, WCET of critical section etc. These parameters are needed by the system integrator to determine the schedulability and timing behavior of the whole system. It also facilitates him in fixing the right duration for a partition window. These parameters can be provided accurately and easily if constant time algorithms are used. For example, the *StartTimer* time management primitive is called inside a critical section from an APEX service. If the primitive is input dependent, then it becomes difficult to estimate the WCET of the critical section or the APEX service.

Although we can find the WCET for an O(n) algorithm, it tends to be too pessimistic and we may reserve a large bandwidth for the service resulting in under utilization of the processor. Thus hard real-time systems do not use heuristic algorithms, as it is difficult to estimate their WCET due to the large number of possible code paths in such algorithms.

4.6.2 Worst-case process release delay

From the time management perspective, worst-case process release delay becomes an important parameter not just to ensure the performance offered to the application by the kernel but also to make the system integrator aware of the release delay faced by a process due to time management. For a high priority process, process release delay becomes an important issue because the process starts executing as soon as it expires. Lower the delay; earlier the process can start executing. We can have a constant process release delay only if the expiry distribution is uniform because as we see that closer expiries can occur only in case of skewed expiry distribution. Primarily, the process release delay is affected by,

a. Expiry distribution such as uniform or skewed distribution. The delay gets affected for closer expiries.
b. The delays due to timer approach such as tick frequency in case of tick based and overshoot delay in case of firm timers.
c. Critical zone length – if the time-out expires inside critical zone. This can be improved by a combined effort of kernel and time management algorithm.
d. Partition event server can be considered as another type of critical zone under the control of the kernel. If a high priority process expires at the beginning of the partition window, then the process cannot start its execution until PES exits. Hence in this case, the algorithm and the code performance of *HandleOutWindowExpiries* primitive becomes critical for the worst-case process release delay.
e. Partition window latency – in case a time-out expires outside its partition window, then the process faces a delay for release until the next partition window.

The performance of all the time management primitives becomes important in one way or the other to reduce the worst-case process release delay. Figure 4.10 shows one possibility of a worst-case process release delay (WPRD). In this figure, process **A** is supposed to get released just inside a half-executed critical section just before the threshold limit (*X*) of the current partition window begins. But since we are nearing partition boundary, the release is deferred until the next partition window. In the next partition window, the incomplete critical section from the previous window is allowed to complete first, followed by partition event

server. The partition event server calls *HandleOutWindowExpiries* that actually inserts process **A** into the ready queue. Therefore the process **A** is released in the next partition window although it expires in its previous window.

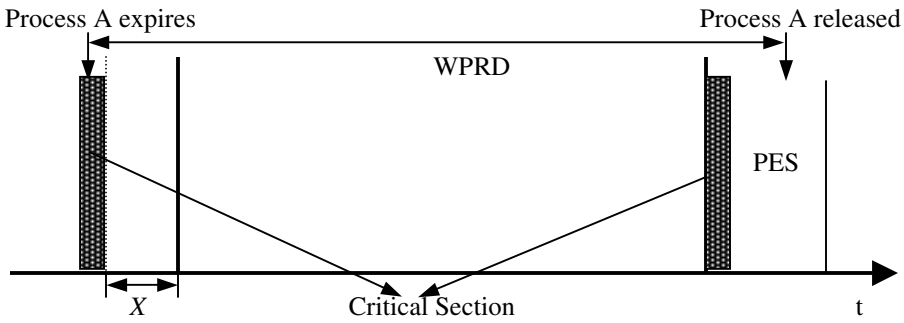


Figure 4.10: A worst-case process release delay

From the timing analysis we understand that the priority of a process has nothing to do with the process release delay. If a low priority process expires at *t1* and a high priority process expires at *t2*, wherein *t1* < *t2*, then the low priority process is released before the high priority process, because the low priority process expires before high priority process. This is also true in case of closer expiries.

4.6.3 Primitive performance

A noted observation done from the above analysis is that to improve process release delay, we need to improve the execution time of all the primitives. It also requires a quicker way of timer reprogramming. Primitive performance is important not just for estimating the WCET of various kernel components but also in reducing the total execution time consumed by time management. Primitive performance is very useful for offline analysis of time management.

4.6.4 Time consumed by time management

Time consumed by time management represents the total time consumed by the time management components out of the total time allocated to a partition for a specified duration. In a partitioned system, wherein the whole system is partitioned, the partition window duration is small. When the partition window duration is small, the time available to run the application becomes critical. If a good time management scheme has achieved O(1) time complexity for its primitives and in achieving so consumes lot of time, then this achievement is not of much use if there isn't much time left for the application to run. Therefore a good

time management scheme must consume least time especially in case of partitioned system, where the partition window itself is small and the time left to run application is little.

The overhead involved in firing a hard timer comprises the interrupt latency. In case of firm timers, overhead also comprises the time taken for a soft timer check and in firing a soft timer. Timer approaches have different overheads and is a useful parameter to observe in relevance with process release delay.

4.6.5 Cache pollution overhead

When an interrupt occurs, the current working set (data and code) of the program in cache may get replaced with the new working set of interrupt service routine. After the ISR completes, the program continues by bringing back its working set to cache from memory, on a need basis. This may result in more cache misses and cache miss penalties. This is called as cache pollution. This affects the worst-case execution time (WCET) of applications and may result in a situation where a process gets affected due to these overheads. Under a stringent deadline system a process might also miss its deadline. Higher the number of interrupts, there could be a higher chance of cache pollution. Most hard real-time systems do not favor processors that have cache due to the unpredictable behavior that may arise due to cache misses.

4.7 Partitioned vs. Non-partitioned system

Time management in partitioned system is different from non-partitioned system primarily due to the partitioning constraints or rules defined in [1]. The partitioning rules puts a restriction on the freedom to handle time-out requests because a partition is not allowed to handle other partition's timer expiries. This means that all processes that are part of inactive expiries can be released only in the partition's next active window, which increases their release delays. But on the other hand partitioned system favors divide and rule strategy for handling time-out requests. This simplifies handling time-out requests by having two components, one for handling current window expiries and the other for handling out window expiries. This naturally reduces the timer queue length if these two components of time-out requests are handled and kept separately. This improves the execution time of primitives. Also since the whole system is partitioned and since partition protection is in place, the time-out requests of each partition are handled independently and to a large extent do not interfere with each other. Non-partitioned system differs in this case and they do not have two components of time-out requests. Thus the efficiency of the algorithm becomes

very critical in non-partitioned system, as the loads on the timer queues are heavy. In case of partitioned system, the total load of the system is automatically distributed.

Having mentioned that, an application developer loses the versatility and freedom to code his application because now the developer needs to be aware of the partition window constraints. If a time-out request expires in a window wherein the respective partition is inactive, then the process faces an inherent release delay. Ensuring strict time partitioning imposes several problems as discussed earlier, which may not exist in a non-partitioned system. We also believe that due to memory and partition window duration constraints, the number of processes n that can be supported in a partition is low. Therefore, n, the input parameter to time management primitives is small in a partitioned system. But in case of a non-partitioned system, there is no restriction on n.

4.8 Summary

This chapter analyzes the timing behavior of real-time application for a partitioned system. It highlights on various issues such as typical time-out requests, harmonic periodic releases, closer expiries etc. and their significance in designing time management module. Analyzing a typical real-time application's timing behavior on a partitioned system helps us in designing better time management algorithms. By understanding the importance of various time related parameters, we can improve the performance of time management module.

Chapter 5

Algorithms under Timer Approaches

This chapter presents algorithms for partitioned system for tick-based and one-shot timer approaches. Few algorithms that are derived from discrete event simulation are modified to suite partition-based environment. The chapter describes the functionalities of time management primitives for these algorithms.

5.1 Requirements

It is very essential that each algorithm implemented for a hard real-time system must meet the industry requirements. Few important requirements to design an algorithm for hard real-time embedded systems are listed down.

1. Verification is an important part of hard real-time systems wherein the system is ensured to have a deterministic behavior. From the operating system perspective it is also required that all system calls have their WCET defined. It becomes apparent that choosing the lowest WCET for a given situation is the best alternative in order to avoid CPU under utilization.
2. In embedded systems, memory is very precious as the whole system (OS + Applications) runs on the memory. It is better if the OS consumes low memory so that enough memory is available for the applications. With this respect, the space complexity of the algorithms used by OS should be low.
3. In a partitioned system, the time is divided to execute the applications. This means that sufficient amount of time within a partition window must be given for the application to run smoothly. Hence the time consumed by the OS must be low. With this respect time complexity of the algorithms used in OS must be low.
4. The algorithms used must have *stable* property. A timer queue that preserves FIFO order on simultaneous expiries is referred to as stable. Simultaneous expiries are expiries that

occur at the same instant of time. Unlike closer expiries, where the expiries are placed close to each other with some offset, simultaneous expiries have zero offset. Among simultaneous expiries, the process that requested the time-out first must be released first. A stable algorithm is often desirable as it may facilitate debugging of the program since all expiries will be evaluated in the order in which they were generated.

5.2 64-bit/32-bit Conversion

The 64-bit absolute time passed to the primitives are converted to 32-bit relative time by subtracting the 64-bit absolute value with the 64-bit start time of the current partition window. This makes 32-bit manipulations more efficient than 64-bit manipulations. It also reduces the space required to store the time-out value in the expiry node. Expiry node is a data structure that holds all the relevant information of a time-out request including forward and backward pointer. Using debugger tool in disassembled mode, we have found that the number of instructions required to manipulate a 64-bit data is higher than a 32-bit data since PowerPC 565 is a 32-bit processor.

A 32-bit relative time-out value cannot be used directly to set a timer for expiry since a 64-bit Time Base register is used in PowerPC 565. While programming the one-shot timer within a partition window, the 32-bit relative time-out is converted back into 64-bit time-out by adding the 32-bit relative time-out to the 64-bit start time of the current partition window.

5.3 StopTimer and HandleTimeOutExpiry Primitives

All the algorithms explained in this chapter uses double-linked list to handle time-out requests. Each node in the linked list is called as an expiry node, which is nothing but a structure of fields. The expiry node is an array of structures indexed by the process ID. Therefore given a process ID, deletion of the node from the double-linked list can be done in O(1) time. Thus *StopTimer* primitive takes O(1) time in all the algorithms described below.

One-shot timer algorithms maintains the linked list of expiries in sorted form and hence *HandleTimeOutExpiry* takes O(1) time to handle a single expiry and re-program the timer for the next nearest expiry. As mentioned in section 4.5.8, the primitive also handles closer expiries deterministically. In case of tick-based approach the primitive handles all the time-out expiries for a given tick width by traversing the linked list and inserting the expired process to the ready queue. The primitive dynamically ensures to exit if the partition boundary is nearing.

5.4 Tick-based Approach

Figure 5.1 shows the details of the tick-based model. The tick interrupts are generated at a certain frequency called the tick frequency. The difference in time between two successive tick interrupts is known as the tick width (in µs). The expiries x, y and z are handled in the tick interrupt **T** irrespective of their actual time of expiry. Precision of tick-based approach is defined by the tick frequency. A low tick width value results in large number of interrupts and thus has high interrupt overhead; but it gives better accuracy. On the other hand, a larger value of tick width has lower interrupt overhead, but provides poor accuracy. Tick frequency = 1 / tick width.

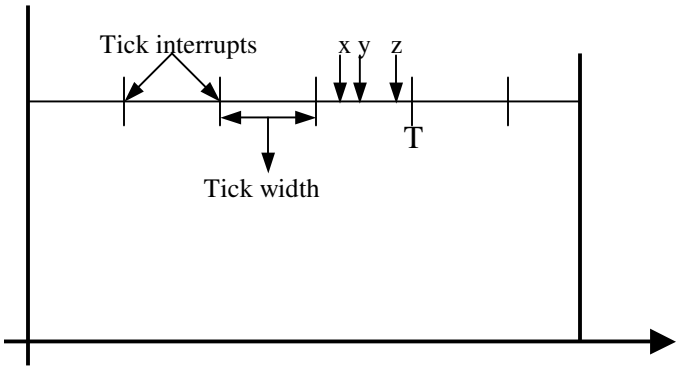


Figure 5.1: Tick-based model

5.4.1 Timing wheel

In a timing wheel scheme we have an array of pointers pointing to a linked list of expiry nodes. The wheel turns one array element for every tick. This is done by incrementing the index of the array on every tick interrupt and checking if the pointer points to a linked list. Time-outs that are supposed to expire in that interval are held in a linked list. The tick ISR handles the expiries in that linked list under a given tick. Figure 5.2 shows the timing wheel implementation for a single partition window.

A global array of size *MAX* named *TA* (for Tick Array) is used to hold the pointers to the head of respective linked list. For the example shown in figure 5.2, there are 8 tick interrupts and hence the first 8 locations of the array are used. If the duration of a partition window divides the window into *T* ticks, then *T* (*T* <= *MAX*) memory locations of the array are used. The same array is re-used for processing expiries for other partitions. But this needs a total of *MAX**4 bytes of memory. Instead of storing the pointers (which require 4 bytes), we can store the unique process ID of the head of linked list. ARINC 653 defines that there can be a maximum of 128 processes under each partition and hence a single byte (8 bits) is enough to

store the process ID (since the process ID ranges from 0 to 127) of the head of linked list. Hence, it would require just *MAX* bytes to store the expiries for the whole system. Head of the linked list is obtained by directly indexing the process ID to the array of expiry nodes.

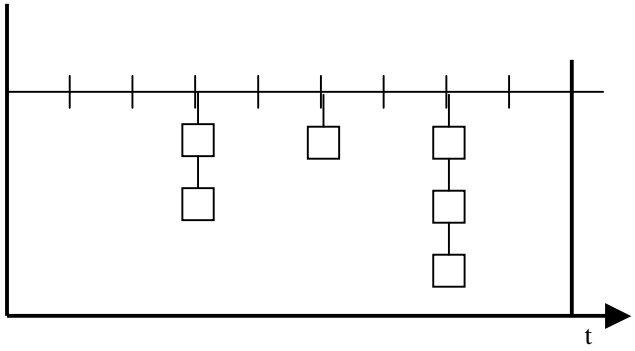


Figure 5.2: Timing wheel for a partition window

5.4.1.1 StartTimer

The *StartTimer* primitive accepts a 64-bit relative time-out. From this we can find the absolute time where the time-out expires by adding the relative time-out to the current system time. If the request expires outside the current window, then the primitive returns by storing the absolute time-out in the process control block. Else, if the time-out expires within the current window, then the equivalent relative value of time-out from the start of the current partition window (CPW) is found. The relative 32-bit time-out value is then converted to the corresponding tick equivalent by dividing the 32-bit relative time-out by the tick width. Then the expiry node of the corresponding request is inserted after the last element of the linked list (tail of the linked list) corresponding to the tick. This ensures that simultaneous expiries are handled in FIFO order and therefore the primitive is stable in handling current window expiries. Insertion takes O(1) time.

Algorithm: StartTimer
Input: Relative Time-out, Absolute End time of current window, Tick width, Absolute Start time of CPW (Current Partition Window)
Output: Inserting the current window expiries into timing wheel

```
Absolute Time-out = Current System Time + Relative Time-out;
If (Absolute Time-out >= Absolute End time of current window)
   (Store the Absolute Time-out in Process Control Block)
   PCB[process_ID] = Absolute Time-out;
Else
```

```
Time-out = Absolute Time-out – Absolute Start time of CPW;
Tick = Time-out / Tick width;
Insert the expiry_node[process ID] to the tail of linked list at TA[Tick];
```

The global variables such as 'Absolute End time of current window' and 'Absolute Start time of CPW' are calculated dynamically in every partition event server. Since the partition schedule is statically determined all these values can be easily calculated from the configuration table.

5.4.1.2 HandleOutWindowExpiries

At the beginning of every partition window, partition event server is invoked in interrupt mode and it calls the primitive *HandleOutWindowExpiries* to handle the time-out requests that have expired before the current partition window (inactive expiries) and insert (or initialize *TA*) the current window expiries. The outstanding expiries (overflow expiries) are not inserted into any overflow buffer list but are kept in process control block (PCB). The PCB stores the absolute time-out of out window expiries. Therefore, *HandleOutWindowExpiries* traverses through all non-dormant processes i.e. PCBs, and checks if any of those has already expired. If so, it inserts the process into the ready queue. Else it checks if the process expires in the current window of the partition in which case it finds the appropriate *Tick* and inserts the expiry node in *TA[Tick]*. Otherwise if the process expires outside the current window, it will remain in PCB and is handled in subsequent partition windows. Here, insertion to ready queue takes O(1) and inserting the expiry node to the timing wheel also takes O(1). Therefore the primitive takes O(n) time to handle *n* expiries. *HandleOutWindowExpiries* involves the comparison of 64-bit absolute time-out and hence the performance of the loop becomes critical. The primitive fails to preserve stable property in handling inactive expiries or in inserting the current window expiries.

Algorithm: HandleOutWindowExpiries
Input: PCB, Absolute End time of current window, Absolute Start time of CPW, Tick width
Output: Handling inactive expiries and inserting the current window expiries in timing wheel.

```
For I = 0 to N
   If (PCB[I] is not dormant)
      If (Absolute Time-out in PCB <= Absolute Current time)
         Insert the process I into the ready queue; (enable it for scheduling)
      Else
```

```
If (Absolute Time-out in PCB < Absolute End Time of current window)
    (Insert the process into timing wheel)
     Time-out = Absolute Time-out – Absolute Start time of CPW;
     Tick = Time-out / Tick width;
     Insert the expiry_node[process ID] to the linked list at TA[Tick];
```

5.4.1.3 Algorithm analysis

The timing wheel algorithm requires *MAX* number of bytes for *TA* and if *M* indicates the maximum number of processes under any partition, then we need O(M) space for expiry node for the whole system. Thus space complexity is *MAX* + O(M). The O(M) space reserved for expiry node can be repeatedly used by each partition (window). Due to this the expiry node data structure has to be initialized in every partition window, which consumes some significant amount of time as part of *HandleOutWindowExpiries*. To avoid this, each partition can have O(N) space where *N* indicate the total number of processes in the respective partition but this increases the total amount of memory consumed by the system.

The performance of *HandleOutWindowExpiries* depends on the out window expiry distribution of the application. Though insertions to ready queue and timing wheel take O(1), it is difficult to make the primitive consume constant time. Since in a worst-case the primitive requires handling *n* expiries, *HandleOutWindowExpiries* cannot be made any better than O(n).

If the current window expiries are already inserted into the timing wheel then we can ease the work of *HandleOutWindowExpiries*, since now we don't have to scan the PCB's for insertion. This is similar to what calendar queue does. The absolute time of expiry can be obtained and hence we can estimate the tick where the time-out expires in a partition window. Therefore we can insert and place these requests in Tick Array. At the time of request we need a way of differentiating the partition window where the time-out expires. But now, insertion to timing wheel takes O(n) since the linked list has to be sorted with respect to partition window. Therefore an advantage obtained for handling out window expiries efficiently can become a disadvantage for *StartTimer* primitive.

The timing wheel algorithm performs in O(1) time which no other algorithms can offer. This helps us to estimate the WCET of critical sections effectively. This is the biggest advantage that the timing wheel algorithm offers. Note that this has been achieved with tick-based approach that has poor timer accuracy.

5.4.2 Hierarchical timing wheel

The algorithm works on the principle of hierarchical timing wheel mentioned in [16]. A major cycle is made up of several partition windows. A time-out request will expire in a major cycle. Within a major cycle, it can expire either in the corresponding partition's active window or in its inactive window. If the time-out expires within a partition's active window, then it would expire in any one of the tick widths. Therefore the first level of hierarchy is the major cycle, the second level is the partition window and the third level is the tick width or the tick where the time-out expires.

Let us assume that a partition can request a time-out that can expire anywhere in the next 10 major cycles. In this algorithm, we have a ten-element array named as *MC* (for major cycle). This array points to an unsorted linked list of expiry nodes of a particular partition that expire in the respective major cycle. The index into this array is modulo incremented in the first partition window of every major cycle. Thus *MC* forms a circular array that points to expiry nodes of a particular major cycle for a particular partition. *MC* array is for the first level of hierarchy.

In the second level of hierarchy, each partition has an array named as *PW* (for partition window), which has elements equal to the number of partition windows in the major cycle for a given partition. For example, if a partition has 4 windows in a major cycle, then the *PW* array has a maximum of four elements. The *PW* array points to unsorted linked list of expiry nodes, which expire in the respective partition window. This array is re-initialized from the linked list of expiry nodes contained in *MC* array in every first partition window of every major cycle, for a particular partition.

As in case of timing wheel, the third level of hierarchy has an array of pointers (named *Tick Array*) to linked list of expiries that expire in a given tick width under a given partition window.

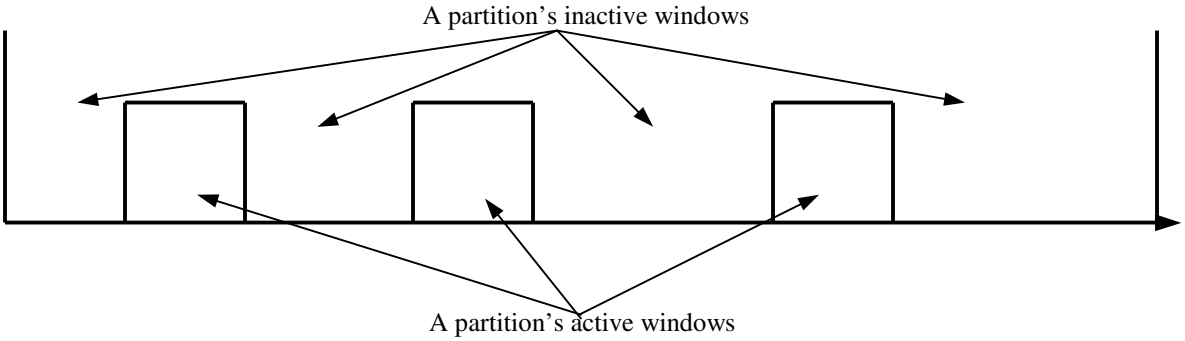


Figure 5.3: Active and Inactive windows of a partition

The time-out requests that expire in a window where the corresponding partition is inactive are called as inactive expiries and are stored in a separate array called *IPW* (for Inactive Partition Window). Each active window of a partition has an inactive window just before the active window of a particular partition. All time-outs that expire in this window are stored in an unsorted linked list pointed by the elements of *IPW* array. Figure 5.3 shows the active and inactive windows of a partition. Note that the first partition window can have one or two inactive windows. If it has two inactive windows, then one would be in the current major cycle and the other is in the previous major cycle.

The expiry node has fields in its structure to store the major cycle, partition window and the tick where the time-out expires. The algorithm fills these fields in the process of handling the time-out request. Figure 5.4 shows the data structures in each hierarchy.

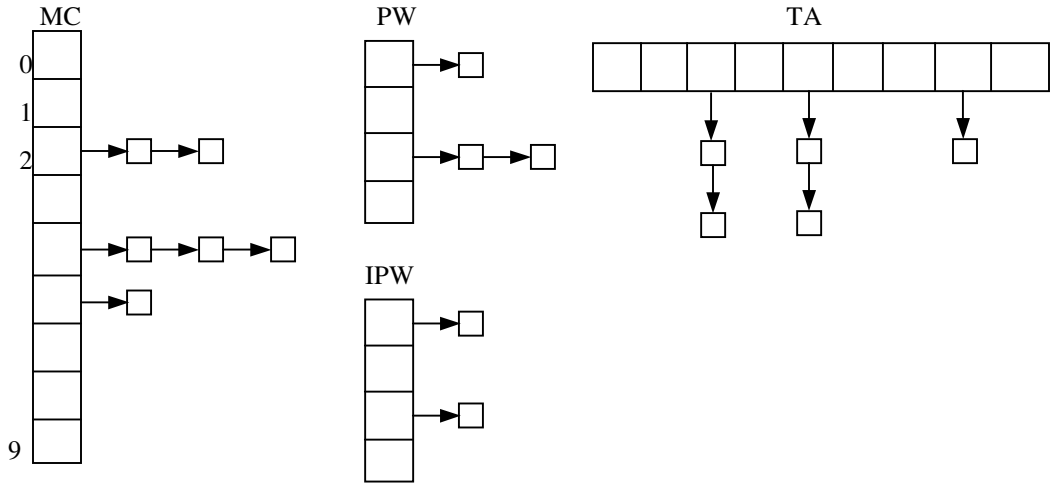


Figure 5.4: Data structures of Hierarchical timing wheel

5.4.2.1 StartTimer

Similar to timing wheel, the *StartTimer* primitive first determines if the new time-out request expires in the current window or outside the current window. If the time-out expires in the current window, the equivalent *Tick* of the corresponding time-out is determined and the expiry node is inserted in the linked list pointed by *TA[Tick]*. Else if the time-out expires outside the current window, then there are two parts in processing the new time-out request. The first part of the algorithm determines the major cycle, partition window in the major cycle and the tick within the partition window where the time-out expires. The second part of the algorithm inserts the expiry node in the linked list pointed by the appropriate array. The absolute time at which the current major cycle starts and the absolute time at which it ends

can be dynamically calculated from the information available in the configuration table. The *StartTimer* primitive is described in the following steps.

1. *Determine the major cycle*
   a. To determine the major cycle, we first check if the absolute time-out expires within the current major cycle. Comparing the absolute time-out with the absolute time where the current major cycle ends can do this. If the time-out expires in the current major cycle, then we set the request's *major cycle* field in the expiry node structure as -1 and we move on to step 2 to determine the partition window where the time-out expires in the current major cycle. A variable named *BTO* (for balance time-out) is set to the value of relative time-out from the start of current major cycle, which is used to determine the partition window in step 2.
   b. Else if the time-out expires outside the current major cycle, then first we subtract a value *T* from the relative time-out value wherein *T* is equal to the difference between the time where the current major cycle ends and the current time. Let us say that the result of this subtraction is stored in *BTO* (for balance time-out). We then check if the *BTO* value is greater than *major cycle width*. If not, the *major cycle* field is set to value 1 to indicate that the time-out expires in the next major cycle. Else if *BTO* is greater than *major cycle width*, then *BTO* is divided by the *major cycle width* to calculate the number of major cycles the expiry spans. The *major cycle* field is stored with a value that is one greater than the result of this division. For example, if the *major cycle* field is 3, then it indicates that the time-out expires in the 3rd major cycle from the current major cycle. The *BTO* is then subtracted with the number of major cycle widths crossed. This resultant *BTO* is used to determine the partition window in step 2. The algorithm given below for determining the major cycle summarizes the above steps.

Algorithm: To determine the major cycle
Input: Relative time-out, major cycle width, Absolute End time of Current Major Cycle,
Absolute Start time of Current Major Cycle
Output: Balance time-out or BTO, Major Cycle where the time-out expires

```
If (Absolute Time-out < Absolute End time of Current Major Cycle)
    Expiry_node. Major_Cycle = -1;
    BTO = Absolute Time-out – Absolute Start time of Current Major Cycle;
Else
    T = Absolute End time of Current Major Cycle – Current System Time;
```

```
BTO = Relative Time-out – T;
Expiry_node. Major_Cycle = 1;
If (BTO > Major_Cycle_Width)
      Major_cycles_crossed = BTO/ Major_Cycle_Width;
      Expiry_node. Major_Cycle = 1 + Major_cycles_crossed;
      BTO = BTO – (Major_cycles_crossed * Major_Cycle_Width);

Return Expiry_node. Major_Cycle;
```

This algorithm takes a constant time to execute and hence the time complexity of the algorithm is O(1). Figure 5.5 shows a better picture on how the major cycle is determined. There are three major cycles shown in the figure and time-out requested by a partition in the current major cycle expires in the respective partition's window in the 2nd major cycle from the current one. In this example, the major cycle crossed by the time-out request is one. Balance time-out (*BTO*) is what finally remains after subtracting the components from the relative time-out.

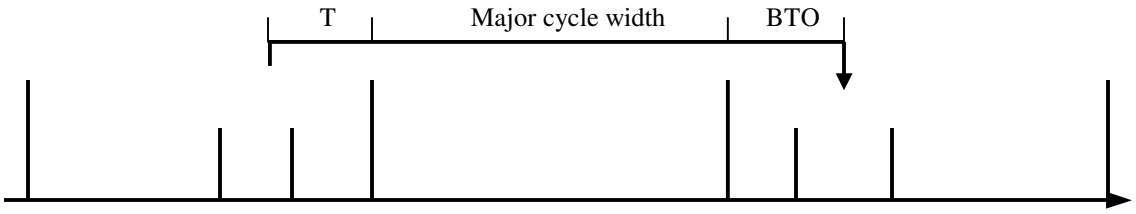


Figure 5.5: Determining the major cycle

2. *Determine the partition window*

The start and end time of each partition window relative to the start of a major cycle is already stored in a table for every partition, named as *PWT* (for Partition Window Table). Given a relative time-out from the start of the major cycle (*BTO*), we can determine the partition window where the time-out expires by searching the entries in the *PWT* table. Let *p* be the number of partition windows in a major cycle for a partition. A *linear search* would require searching if *BTO* falls within the start and end time range in each entries of *PWT* (i.e. for each partition window). The time complexity of this approach is O(*p*). Instead we can also apply the *binary search* technique since the entries in *PWT* are sorted. This makes the complexity of this step to be O(log*p*), hence a constant. If the search fails, then it means that the time-out expires in an inactive partition window and hence needs to be handled as inactive expiries. The *partition window* field is set to the partition window number where the search succeeds

or fails. Inactive expiries are differentiated from active window expiries by setting the most significant bit (MSB) of the *partition window* field. In this step, if the time-out expires in the active window, then *BTO* is subtracted with the start time of the corresponding partition window where the time-out expires. This *BTO* is used in step 3.

3. *Determine the tick*

   The tick can be determined similar to the idea mentioned in timing wheel algorithm. The *BTO* obtained from step 2 is divided by the tick width to obtain the tick that is also stored in the expiry node structure. This step also takes a constant time.

4. *Inserting the time-out request in the array*

   Since the linked lists are unsorted, insertion takes O(1) time. The processing of the linked list is always done from the head of linked list. Once we obtain the fields of expiry node from the above steps, the expiry node is inserted into the linked list pointed by the appropriate array. We know that if a time-out expires within the current window, then the expiry node is inserted at *TA[Tick]* for the appropriate *Tick*. If the time-out expires within the current major cycle but in a different partition window of the same partition, then expiry node is inserted in to the linked list pointed by the *PW* array for the respective partition window. If a time-out expires in a major cycle other than the current one, then the expiry node is inserted into the linked list pointed by the respective major cycle entry of the *MC* array.

   The far future expiries stored in *MC* array are moved to *PW* and *IPW* array when the respective major cycle becomes active. *HandleOutWindowExpiries* called from PES does this in every first partition window for a particular partition.

Algorithm: StartTimer

Input: Relative Time-out, Absolute End Time of current window, Absolute Start Time of CPW.

Output: The expiry node is inserted in the linked list of appropriate array.

```
Absolute Time-out = Relative Time-out + Current System Time;
If (Absolute time-out < Absolute End Time of current window)
    Time-out = Absolute Time-out – Absolute Start Time of CPW;
    Tick = Time-out / Tick width;
    Insert the expiry_node[process ID] to the tail of linked list at TA[Tick];
Else
    m_c = Determine the Major cycle;
    p_w = Determine the Partition window;
```

```
tick = Determine the tick;
If (m_c == -1)
    If (MSB of p_w == 1)
        Reset MSB of p_w;
        Insert the expiry_node[process_ID] in the linked list at IPW[p_w];
    Else
        Insert the expiry_node[process_ID] in the linked list at PW[p_w];
Else
    m_c = (MC_index + m_c) mod 10;
    Insert the expiry_node[process_ID] in the linked list at MC[m_c];
```

The overall time complexity of *StartTimer* is O(log*p*) where *p* is the number of partition windows in a major cycle for a particular partition. For a given partition, *p* is constant and hence log*p* is also constant. Therefore *StartTimer* primitive consumes a constant time.

5.4.2.2 HandleOutWindowExpiries

The primitive initializes the index into the *MC* array to 0 in the first major cycle and is then modulo (mod 10) incremented in the first partition window of every major cycle. It is the responsibility of *HandleOutWindowExpiries* to move the expiry nodes between the arrays (*MC*, *PW*, *IPW* or *TA*). The first partition window in every major cycle for a particular partition moves the expiry nodes from the respective entry in the *MC* array to the respective entries in the *PW* and *IPW* array (if any) for that major cycle. *HandleOutWindowExpiries* can determine if the current partition window is the first partition window of a major cycle from the information present in configuration table. The index into the *PW* and *IPW* array is initialized to 0 in the first partition window in every major cycle and is then incremented in the subsequent partition windows.

If there is an expiry node that is due to expire in the first partition window itself, then it is directly linked in *TA[tick]*. The information on which partition window (active or inactive) a time-out expires in the current major cycle is directly obtained from the structure fields of the expiry node. This is used to index the *PW* and *IPW* array.

This primitive checks for any time-out requests that has expired before the current window. Inspecting the appropriate entry in the *IPW* array does this. The primitive handles all such inactive expiries by traversing the linked list and inserting the expired process to the ready queue. In the absence of such inactive expiries, the algorithm can simply move to the next step. Note that inactive window for the first partition window of a major cycle may span from previous major cycle to the current major cycle. The inactive expiries of the previous

major cycle can be handled by scanning the last entry of the *IPW* array in the first partition window of a major cycle, for a partition.

When the respective partition window becomes active, the expiry nodes linked for that partition window in *PW* array is inserted into *TA[tick]*, wherein the tick value is also obtained from the structure fields that has been pre-calculated in *StartTimer* primitive.

The time complexity of this primitive is O(n) where *n* indicates the number of time-out requests. This is because in the worst case the traversal requires a total of de-linking and linking *n* expiries.

Algorithm: HandleOutWindowExpiries
Input: MC, PW, IPW and TA array, MC_index, PW_index and IPW_index
Output: Moving the expiry nodes from a source array to a destination array appropriately and initializing the index. Handling expiries by inserting them to ready queue.

```
If (CPW is the first window in a major cycle)
    { Handle inactive expiries from previous major cycle, if any }
    While (IPW[last entry] is non-nil)
        Handle the inactive expiries by inserting the process to ready queue;
    MC_index = (MC_index + 1) mod 10;
    PW_index = 0;
    IPW_index = 0;
    While (MC[MC_index] is non-nil)
        Traverse the linked list and move the expiry_nodes from MC[MC_index] to respective
        entries in PW and IPW array;
        If (a time-out expires in current window)
            Insert the expiry_node[process_ID] in the linked list at TA[tick];
    While (IPW[first inactive window] is non-nil)
        Handle the inactive expiries by inserting the process to ready queue;
Else
    PW_index = PW_index + 1;
    IPW_index = IPW_index + 1;
    While (IPW[IPW_index] is non-nil)
        Handle the inactive expiries by inserting the process to ready queue;
    While (PW[PW_index] is non-nil)
        Traverse the linked list and move the expiry_nodes from PW[PW_index] to the respective
        linked list at TA[tick];
```

5.4.2.3 Algorithm analysis

Hierarchical timing wheel algorithm requires more memory space than timing wheel algorithm. In our example considered above, each partition requires a total of 10 (for *MC*) + 2*p* (for *PW* and *IPW*) bytes of memory. The *TA* used to handle expiries within a partition window can be reused for other partition windows (of other partitions).

By comparing this algorithm with timing wheel algorithm, we understand that this algorithm is better if we have more number of out window expiries that expire in far future (across major cycles). The advantage is primarily observed in case of *HandleOutWindowExpiries* since the Hierarchical timing wheel algorithm need not have to traverse the entire PCB to handle the expiries each time as in case of timing wheel. But the worst case complexity of this primitive is still O(n). The algorithm also provides stable property in handling all forms of expiries, which timing wheel doesn't.

As we observe from *HandleOutWindowExpiries*, PES of the first partition window in a major cycle consumes more time than PES of other partition windows. This is because the primitive has to move the expiries from *MC* array to the respective entries in the *PW* and *IPW* array of all partition windows. Instead the partition schedule can be framed in such a way that the first window of a major cycle belongs to a kernel partition (usually named as system management partition) that performs the moving activity for all the partition windows (including that of other partitions). This kernel partition performs the kernel activities for all the partitions or applications and hence helps in reducing the WCET of PES.

The insertion time of *StartTimer* primitive (O(log*p*)) in this approach is slightly higher than timing wheel. The time and space required to maintain hierarchical timing wheel is higher than timing wheel.

5.4.3 Summary

In tick-based approach, the tick width now becomes part of the worst-case process release delay in addition to other latencies. This delay is the inherent part of the approach. If a tick interrupts at a time when the kernel is in critical section, then the process will face further delays by the WCET of the critical section.

$$\textit{Worst-case Process Release Delay}^{\textit{Tick-based}} = \textit{Tick Width} + \textit{Process Release Delay}$$

Advantages

1. Tick based approach is very simple and has a deterministic insertion and deletion time of O(1). This also improves the performance of *HandleOutWindowExpiries*.

2. Different applications can choose different tick frequency that suits their requirement; hence various factors can be controlled with individual verification analysis.
3. Timer reprogramming overhead is nil.
4. Having a constant-time insertion and deletion favors an accurate estimation of critical section since *StartTimer* and *StopTimer* primitive are called inside critical sections. Further this also helps in estimating the WCET of APEX service.

Disadvantages

1. This approach has low accuracy as the process has to wait for a worst-case delay governed by tick frequency.
2. This approach pollutes the cache too often and affects the WCET predictability of the application. Hence it has considerable overhead.
3. Coming up with an optimum tick frequency is a major challenge. Having a high frequency increases the accuracy and also the interrupt overhead. On the other hand reducing the frequency gives a low overhead but will have a poor accuracy.

5.5 One-shot timer approach

In this section we present few time management algorithms to handle time-out requests using one-shot timer for partitioned system. Notably, there have been few algorithms in this approach and this section presents new techniques on how discrete event simulation algorithms can be used with one-shot timer for real-time systems.

Following are the desirable properties for one-shot timer algorithms.

- It is better if the time-out requests are kept sorted. This reduces the timer-reprogramming overhead and handles closer expiries in a better way. Else there should be a faster way of finding the next nearest expiry. Therefore the insertion and deletion must ensure that the list of time-out expiries remain sorted.
- Since the deletion of any time-out request is equally likely to happen, it would be better if we could do this with least time complexity. Double-linked list can perform deletion better than tree or array based data structure. Since the expiry node is a structure with different fields, shifting or swapping a structure consumes more time. From our experiments, we have observed that algorithms based on array perform poorer due to the shifting and maintenance overhead, compared to simple linked list algorithms.

We have tried using the obvious linked list algorithms that suits the requirement. Linked list based algorithms take O(n) for insertion and, O(1) for deletion and processing expiry. Three

algorithms are discussed below. They are Dual Linked Lists (DLL), Hierarchical Multiple Linked Lists (HMLL) and Bit-map Calendar Queue (BCQ). All these algorithms use sorted double-linked list. Multiple linked lists (MLL) is an algorithm similar to HMLL except that out window expiries are handled in a naïve way like timing wheel discussed earlier. DLL has behavior similar to that of [38] and variants of multiple linked lists (MLL) have been proposed in [39] for tick-based approach. However in our model MLL is realized with bit-map manipulations for one-shot timer. All the algorithms described in this section ensure stable property.

5.5.1 Dual Linked Lists

This algorithm has two sorted linked lists maintained for each partition (hence the name dual linked lists). The primary linked list stores the time-out requests that expire in the current window and the secondary linked list stores the time-out requests that expire out of the current window. Both the linked lists have expiry nodes that are kept sorted. Each linked list has a pointer pointing to the head of the linked list. Insertion to primary linked list is done by scanning the list from the end, which exploits the property mentioned in section 4.5.2. Searching the list from the beginning does insertion to secondary linked list.

When the current window ends, the expiries in the primary linked list would have been handled and the primary linked list becomes empty. In the partition's next window, *HandleOutWindowExpiries* divides the secondary linked list into three parts, on a need basis.

1. Time-outs that have expired before the current time will be handled by inserting the processes into the ready queue.
2. Current window expiries are marked as the primary linked list and the timer is programmed to the time-out value present in the head of the linked list.
3. Out window expiries are marked as the secondary linked list.

Figure 5.6 shows on how the secondary linked list is broken into three components. As we see from the figure, part of the secondary linked list becomes primary linked list and the other part remains in secondary linked list. This requires the traversal of the linked list to have a pointer to the head of secondary linked list.

Normally the time-out requests that expire within the current window are converted to their equivalent 32-bit relative time-out from the start time of current partition window. This is to enable the optimization of having 32-bit time manipulations instead of 64-bit. In timing wheel out window expiries are stored as absolute time-out in 64-bit variable. This can cause problems for time manipulations in *HandleOutWindowExpiries* primitive that consumes more time for 64-bit manipulations. DLL solves this problem by storing only 32-bit relative

time-outs and *HandleOutWindowExpiries* dynamically updates the time-outs present in each node of the linked list.

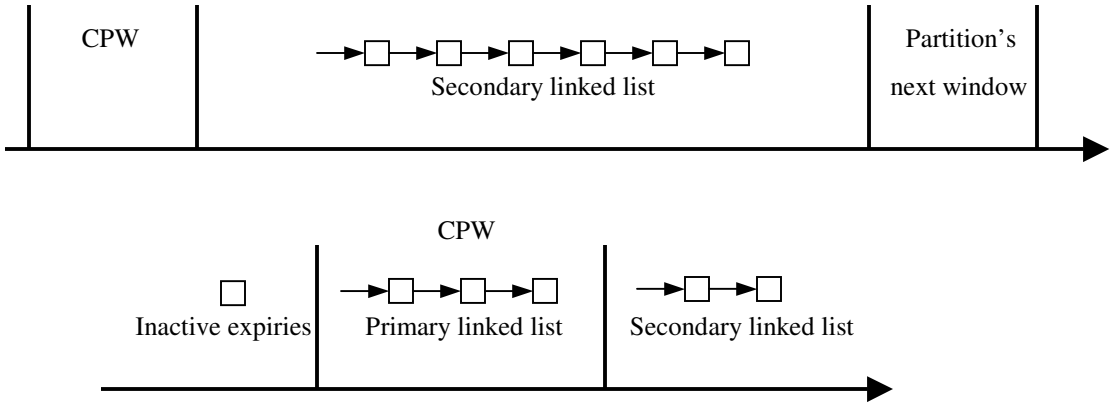


Figure 5.6: Dual linked lists

5.5.1.1 StartTimer

If the time-out expires within the current window, then the request is inserted in its sorted place in the primary linked list by scanning the linked list from its end. If the time-out expires out of the current window, then the absolute time-out is converted to 32-bit relative time-out by subtracting the absolute time-out with the absolute end time of the current window. This request is then inserted in its sorted place in the secondary linked list by scanning the linked list from the beginning. All out window expiries are inserted with a time-out relative to the current window end. *HandleOutWindowExpiries* further manipulates the out window time-outs in the partition's next window. The *StartTimer* primitive takes O(n) time to insert the new request in its proper place in a sorted linked list. With two sorted linked lists we expect to improve the average time of insertion because the load is shared between the lists. It is assumed that the relative time-out value passed to *StartTimer* primitive does not cross the 32-bit limit.

Algorithm: StartTimer
Input: Relative Time-out, Absolute End time of current window, Absolute Start time of CPW
Output: Inserting the new time-out request in primary or secondary linked list

```
Absolute Time-out = Relative Time-out + Current System Time;
If (Absolute Time-out >= Absolute End time of current window)
    New relative Time-out = Absolute Time-out – Absolute End time of current window;
    Insert the new request in secondary linked list by scanning the list from beginning;
```

Else

New relative Time-out = Absolute Time-out – Absolute Start time of CPW;

Insert the new request in primary linked list by scanning the list from end;

5.5.1.2 HandleOutWindowExpiries

The primitive traverses and handles expiries in the secondary linked list. It also forms a new primary linked list by breaking the secondary linked list, on a need basis. This is because, if there are no current window or out window expiries, then the breakdown will not happen and a new linked list gets appropriately created in the *StartTimer* primitive. This primitive has three parts and each part is executed depending on the expiries present in individual category.

1. The first part of the primitive checks for any time-outs that has expired before the current time, starting from the head of the secondary linked list. Since the linked list is sorted based on the relative time-outs that are relative to the previous window end of the partition, we can easily compare and find out for such expiries. The current system time is also made relative to the previous window end of the partition.
2. The second part of the algorithm traverses the linked list to separate the primary linked list. While traversing it adjusts the relative time-outs in each node relative to the start time of current partition window. It also programs the timer for the time-out that is in the head of primary linked list. *HandleOutWindowExpiries* is called from PES that executes in interrupt mode and therefore interrupts are disabled until PES exits. If a time-out expires before PES exits, then it remains as pending interrupt and the expiry is immediately processed as soon as PES exits. When the second part finishes the traversal, a new primary linked list would have formed.
3. The third part of the primitive forms a new secondary linked list for any out window expiries. Here the primitive traverses the secondary linked list and adjusts the relative time-outs in each node relative to the end time of current window.

Figure 5.7 explains how the adjusted relative time-outs preserve the sorted property. Figure 5.7 shows two partition windows (**A** and **B**) of the same partition. When the partition window **A** finishes, the secondary linked list has expiries **E1**, **E2** and **E3**. When the control reaches PES in partition window **B**, *HandleOutWindowExpiries* handles the expiries that occurred before the current time, i.e. **E1** in this case. Thus the process that has expired at **E1** is inserted into the ready queue. **E2** expires in the current window and hence **E2** is a node in the newly formed primary linked list. **E3** expires outside the current window and hence **E3** is a node in the newly formed secondary linked list. As time progresses, new time-out request from *StartTimer* primitive that expires in the current window are inserted into the primary linked

list and these time-out requests are relative to the start time of the current partition window as in case of **E2**. Similarly new time-out requests that expire outside the current window are made relative to the current window end and inserted into secondary linked list. Thus the adjustments ensure that the linked list remains sorted.

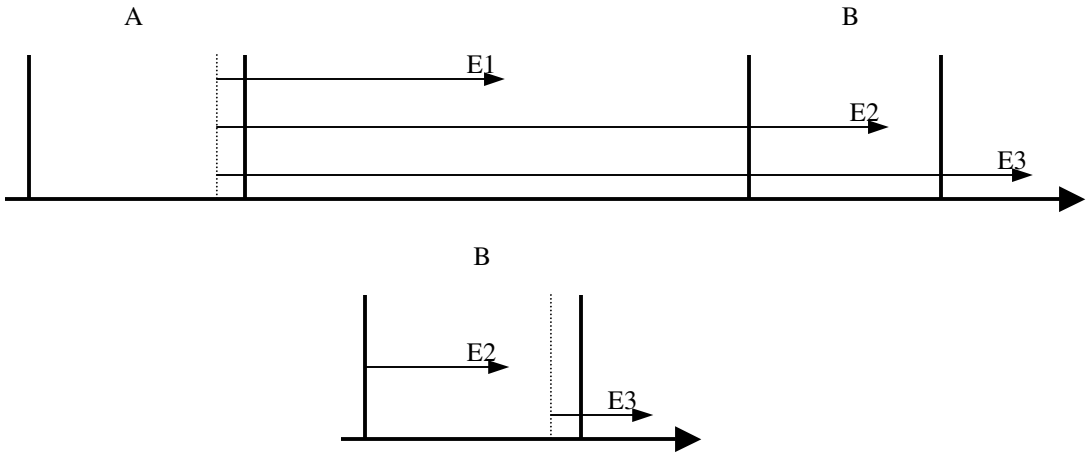


Figure 5.7: Relative time-outs

The worst-case time complexity of this primitive is O(n) since the worst-case length of the linked list is *n* and the primitive traverses each node in the linked list. But on an average, *HandleOutWindowExpiries* performs better since the algorithm avoids scanning all PCBs looking for expiries as in case of timing wheel.

Algorithm: HandleOutWindowExpiries
Input: Absolute End time of partition's previous window (PPW), Absolute Start time of CPW, Absolute End time of current window.
Output: Breakdown of secondary linked list into primary and secondary

```
Current time = Current System Time – Absolute End time of PPW;
Relative End time of current window = Absolute End time of current window –
                                      Absolute End time of PPW;
Relative offset = Absolute Start time of CPW – Absolute End time of PPW;

Head = Head of secondary linked list;
While (Head is non-nil AND Head->time-out <= Current time)
        Handle expired nodes by inserting the process to ready queue;
        Head = Head->next;
```

```
Head of primary linked list = Head;
{Break the secondary linked list to form a new primary linked list}
While (Head is non-nil AND Head->time-out < Relative End time of current window)
        Traverse and adjust the time-out relative to the start of CPW;
Program the timer for the head of primary linked list;

Head of secondary linked list = Head;
{The nodes left forms the secondary linked list}
While (Head is non-nil)
        Traverse and adjust the time-out relative to the end time of current window;
```

5.5.1.3 Algorithm analysis

All the partitions in the system have two linked lists to manage time-out expiries. Hence this algorithm requires O(N) space per partition, wherein N indicates the total number of processes in the respective partition.

This algorithm works on the principle of divide and rule and is based on having two separate linked lists to handle the expiries. Therefore an insertion requires the traversal of nodes only in the respective linked list. Hence we can expect that the average case performance of this algorithm is better. Since the out window expiries are already stored as sorted in secondary linked list, it avoids any new insertions in *HandleOutWindowExpiries* as done in timing wheel. On a 64-bit processor, the linked list can store the 64-bit absolute time-out value since the processor supports 64-bit manipulations. In such a case, *HandleOutWindowExpiries* do not have to traverse the secondary linked list for adjustments and therefore the primitive can perform still better.

5.5.2 Hierarchical Multiple Linked Lists

This algorithm primarily derives its nature from hierarchical timing wheel. In timing wheel, the tick frequency naturally divides the partition duration into equal widths. This enables constant-time insertion of expiries into their respective linked list. The algorithm presented here realizes a similar method for one-shot timer with higher accuracy.

In hierarchical multiple linked lists (HMLL) algorithm, the partition window is divided into equal sized widths called slots that are in powers of 2 (like 64, 32 etc). Each width is called as a slot (S). Each slot maintains a sorted double-linked list for time-outs that expire under a slot. A header points to a linked list and hence multiple headers points to multiple sorted linked lists. This is a way of dividing a single sorted linked list into multiple sorted

linked lists with multiple headers. This reduces the average insertion time although the worst-case insertion time is still O(n), i.e. when all the requests expire in the same slot. Figure 5.8 shows the slots for a partition window that is divided into 64 slots.

The hierarchy part of the algorithm is similar to hierarchical timing wheel. That is, the first level of hierarchy is the major cycle, the second level is the partition window and the third level is the slot within the partition window where the time-out expires. If the time-out expires in a window where the partition is inactive, then those time-out requests are maintained and handled similar to hierarchical timing wheel. The third level of hierarchy has an array called slot array (*SA*) for slots within a partition window. The scanning of the linked list for insertion into linked list pointed by *SA* begins from the head of linked list.

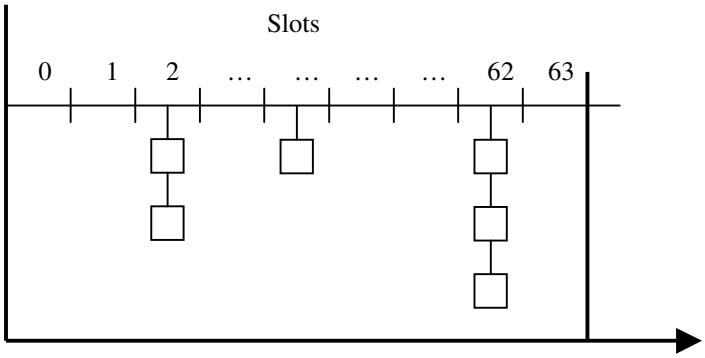


Figure 5.8: Multiple linked lists

5.5.2.1 Bit-map manipulations

From figure 5.8, we understand that to begin with, the timer is programmed with the time-out present in the head of linked list of the first nearest slot that has a linked list. Once the timer expires, it has to be reprogrammed to expire at the time-out value present in the next node of the linked list. If it doesn't have a next node (in case there was only one expiry in that slot), then the timer has to be reprogrammed with the time-out of the head of linked list pointed by the next nearest slot. Finding the next nearest slot that has an expiry is difficult. A straightforward approach would be to scan all the slots from the next slot until we find a slot that has a linked list. This clearly consumes lot of time. We use bit maps to find the next nearest slot that has the linked list. All operations on this bit-map takes O(1) time. Let us assume that partition window duration is divided into 64 slots. The bit map has the following components.

***Slot Array (SA) of 64 pointers to linked list*:** This array points to sorted double-linked list of expiry nodes. Each array element points to a linked list of expiry nodes that expire in the corresponding slot. The head of the linked list has the first expiry to occur in that slot.

***Bit-map*:** Whenever a process requests for a time-out that expires within the partition window, the corresponding time slot *S* of that expiry is determined. Dividing the relative time-out request with the slot width can do this. The slot width is obtained by dividing the partition window duration with 64. Then the bit-map is updated by setting the corresponding S^{th} bit in a 64-bit variable. Similarly when an expiry node is deleted and if the linked list corresponding to that slot is empty, then we reset the corresponding S^{th} bit in the 64-bit variable. This 64-bit variable is divided in to eight 8-bit bytes in the form of a byte array with 8 elements named as *SE* (for slot entry). We have another 8-bit byte named *SG* (for slot group) whose bit positions indicate which element of *SE* array is set. That is, if the byte in *SE*[2] has any one of its bits set, then the 2^{nd} bit of *SG* is set. If *SE*[2] = 0, then the 2^{nd} bit of *SG* is 0. This data structure setup is shown in figure 5.9.

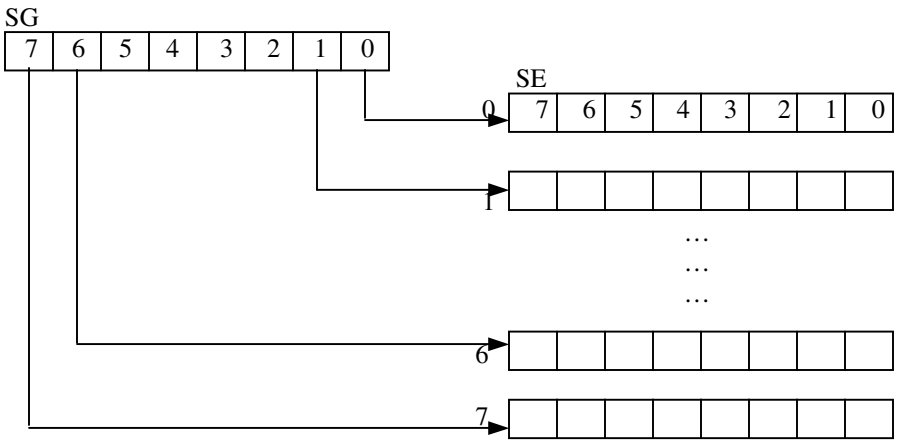


Figure 5.9: Bit-map data structure

***Constant Map Table*:** This is a 256-byte constant array named as *MT* (for map table) that contains the bit position of the least significant bit set for a given 8-bit value. For example, if the value is 32, the bit position of the least significant bit set is 5 and therefore the constant array *MT*[32] = 5. Similarly *MT*[12] = 2 (12 = 1100). An 8-bit variable can take up to 256 values and hence a constant array of size 256 bytes is used to hold the bit position of the least significant bit set. Indexing the 8-bit value directly into this array gives us the bit position of the least significant bit set for that value. This table is only referred and is not updated, hence is a constant array. A brute force approach would be to scan all the bit positions to find the

bit position of the least significant bit set, which takes time proportional to the number of bits. Whereas with a map table we can always find the LSB set in O(1) time.

***Bit-map Algorithm*:** Whenever a process requests a time-out, we can determine the slot where it expires by simple calculations that take O(1) time. Then we insert the expiry node in the linked list of that slot by traversing the linked list to find a proper position in the sorted linked list. This takes O(n) if all the requests expire under the same slot. After this we need to update the bit-map variables *SE* and *SG* to reflect the insertion. The slot ranges between 0-63, 0 indicates the leftmost slot (at the start of the partition window) and 63 indicates the rightmost slot (towards the end of the partition window). This is shown in figure 5.8.

Given a slot value in an 8-bit variable *S*, bit position 5, 4 and 3 indicate the *SE* array element that has to be set. Bit 2, 1 and 0 indicate the bit position in that *SE* array element that has to be set. Bit 7 and 6 of the slot variable *S* is ignored. For example, if the slot value is 43, then it can be represented as 00101011 in the binary format. Of these bits, bit 5, 4 and 3 has 101 i.e. value 5 and hence *SE*[5] is used. Bit 2, 1 and 0 has 011 i.e. value 3 and hence the 3rd bit in *SE*[5] is set. After this the 5th bit in *SG* is set to indicate that *SE*[5] is non-zero.

Each time a time-out request is inserted to its slot in the linked list, the corresponding bit in *SE* and *SG* is updated to reflect the insertion. Similarly each time an expiry node is deleted from its linked list of the corresponding slot and if the linked list is empty for that slot, then we reset the corresponding bits in *SE* and *SG* to reflect that there are no expiries in that slot. All these operations take O(1) time. The operation is,

SE[*S*(5:3)] = *SE*[*S*(5:3)] OR (1 << *S*(2:0));
SG = *SG* OR (1 << *S*(5:3));

The bit-map and the constant map table data structure are basically maintained to facilitate in finding the slot of the next nearest expiry once a linked list of a slot becomes empty. In other words, it is maintained to ease timer reprogramming, since the head of the linked list in the next nearest slot will be the next time-out to expire. Thus timer reprogramming can be done in O(1) time.

The next nearest/closest expiry can be found from the least significant bit set in the 64-bit variable. This variable is represented as *SE* array with eight elements. This array is governed by *SG* variable. The least significant bit set (say X) in *SG* gives the index into the *SE* array that has one of its bits set. The least significant bit set (say Y) in that array element (*SE*) gives us the slot of the next nearest expiry. The least significant bit set of any 8-bit value can be obtained by directly indexing the constant map table. The slot value of the next nearest

expiry is the value obtained by concatenating X and Y i.e. X is left shifted by 3 and is OR'd with Y. The steps are as follows:

X = *MT*[*SG*];
Y = *MT*[*SE*[X]];
S = (X << 3) OR Y;

Head of the linked list at *SA*[S] is the next nearest time-out request that has to be programmed. Therefore this operation also takes O(1) time. Therefore all the operations are done efficiently in just O(1) time. A similar bit-map can also be implemented for other powers of 2 (32, 16 etc.).

Let us now try to understand these bit-map operations with an example. Figure 5.10 shows the picture of this example for partition window duration of 20ms. The duration is divided by 64 and each slot has a width of 312µs. Let us assume that the partition window starts from an absolute time of 0µs. The figure shows three time-out requests, the first one expires at 1900µs, the second expires at 2600µs and the third expires at 2650µs. The first request of 1900 is linked in the slot 6 (1900/312 = 6) and the other two requests are linked in the slot 8 (2600/312 = 8).

For the example considered in figure 5.10, *SG* = 00000011, *SE*[0] = 01000000 and *SE*[1] = 00000001 in binary format (please refer figure 5.8). After processing the time-out request for 1900, bit 6 of *SE*[0] is reset and since *SE*[0] has a value of 0 (after resetting bit 6), bit 0 of *SG* is also reset. That is now, *SG* = 00000010 and *SE*[0] = 00000000. The timer now has to be reprogrammed with the next nearest expiry that is in slot 8. The value 8 is obtained as follows:

X = *MT*[*SG*] → X = *MT*[2] = 1
Y = *MT*[*SE*[X]] → Y = *MT*[*SE*[1]] = *MT*[1] = 0
S = (X << 3) OR Y → S = (1 << 3) OR 0 = 8

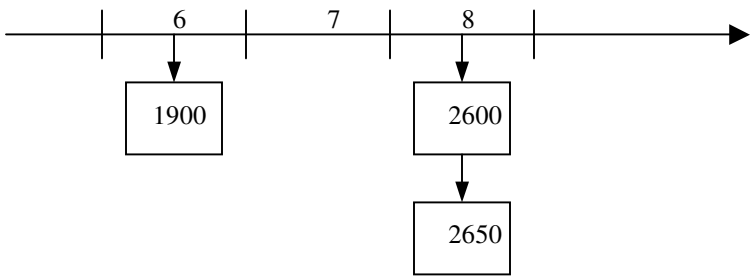


Figure 5.10: Bit-map example

5.5.2.2 StartTimer

The hierarchical part of the algorithm is same as that of hierarchical timing wheel algorithm described in section 5.4.2 and hence takes constant time.

If the time-out expires within the current window, then the request is inserted into the sorted linked list at corresponding slot in *SA*. If the time-out expires outside the current window then the primitive follows the same algorithm as that of hierarchical timing wheel to find the respective major cycle, partition window and the slot. The time-out request is inserted in the respective array. If the new time-out request happens to be the nearest expiry, then the timer is reprogrammed. The bit-maps are appropriately updated.

The worst-case performance of the algorithm is O(n) (for skewed distribution) but on an average we may expect that this primitive performs better if the expiries are uniformly distributed and if the total number of time-out requests is less than 64. The algorithm ensures stable property while inserting simultaneous expiries.

Algorithm: StartTimer
Input: Relative time-out, Absolute End time of current window, partition window duration
Output: Inserting the time-out request in the linked list of the appropriate array, updating the bit-maps and programming the timer if needed.

```
Absolute Time-out = Relative Time-out + Current System Time;
If (Absolute time-out < Absolute End Time of current window)
    Time-out = Absolute Time-out – Absolute Start Time of CPW;
    Slot width = Partition window duration / 64;
    Slot = Time-out / Slot width;
    Insert the expiry_node[process ID] to the linked list at SA[Slot];
    Update the bit-maps and program the timer appropriately;
Else
    Determine the major cycle, partition window and slot as mentioned in section 5.4.2.1;
    Insert the request in the linked list of appropriate array; (MC, PW, IPW or SA);
```

5.5.2.3 HandleOutWindowExpiries

The first partition window in every major cycle moves the expiry nodes from one level to the next lower level of the hierarchy as in case of hierarchical timing wheel. In every partition window, *HandleOutWindowExpiries* moves the expiry nodes from *PW* array to *SA* if required. This involves the insertion of time-out requests into the corresponding linked list

under the slot array which takes O(n) time. All inactive expiries are handled by referring the *IPW* array. This primitive has a worst-case $O(n^2)$ time complexity. But if the average case length of the slot's linked list is a constant, then this primitive has a time complexity of O(n). The algorithm is similar to what is given in section 5.4.2.2, except that here the bit-maps are updated appropriately.

5.5.2.4 Algorithm analysis

This algorithm requires memory of 9 bytes for bit-maps and 64 bytes for slot array (*SA)*, which is roughly proportional to the 2's power chosen. It requires an additional 256 bytes of memory for constant map table for the entire system. Apart from this it requires a total of 10 (for *MC*) + 2*p* (for *PW* and *IPW*) amount of memory space per partition, where *p* indicates the number of partition windows in every major cycle for a particular partition. Expiry nodes require O(N) space per partition where *N* is the total number of processes in the respective partition. Hence this algorithm consumes more memory than other algorithms i.e. O(N + *C*), where *C* is the constant number of bytes required for bit-maps and arrays.

Hierarchical multiple linked lists algorithm attempts to reduce the insertion time as well as handles out window expiries intelligently. As in case of Hierarchical timing wheel, the algorithm performs better when there are huge numbers of far future expiries. The algorithm needs to maintain bit-maps in every operation which takes O(1) time.

As the slot width tends to 0, the length of a linked list under a slot also tends to 0 and hence the insertion time reduces. The performance of *StartTimer* primitive depends on the partition window duration. Larger partition window duration results in a larger slot width given that 64 is a constant. This can be improved if we choose 128 instead of 64, but then bit-map updations for 128 consume more time. If the slot width is large, then the probability that it would have more number of expiries is high. Thus the length of the linked list for a slot will increase and hence the insertion time will also increase. If the partition window duration is divided by 8, then we can have a simpler implementation of the bit-map algorithm but the slot width increases and therefore the average insertion time. Another advantage in maintaining a hierarchy is that far future expiries are stored in unsorted manner and hence doesn't incur the cost of sorted insertion. This will have a greater advantage if far future expiries have a higher tendency to get deleted.

5.5.3 Bit-map Calendar Queue

Bit-map calendar queue works on the calendar queue principle of inserting far future active window expiries into the determined (i.e. corresponding) slot's sorted linked list along with

current window expiries. The hierarchical information of the expiry obtained is stored in expiry node and is used to maintain the slot's linked list sorted. Each slot's linked list is sorted by major cycle, partition window and the time of expiry within the slot. Thus each partition has 64 pointers to linked list of expiries that hold both current window and future active window expiries. Just as in case of calendar queue, no overflow list is required to maintain future active window expiries, however inactive expiries are maintained in overflow list just like hierarchical timing wheel. This is inevitable since the 64 slots divide only the active partition window. The algorithm works even if partition has different window sizes.

The algorithm requires re-initializing bit-maps in every partition window (done in *HandleOutWindowExpiries*), which in turn requires the traversal of 64 slots to determine which of the already inserted time-out requests expire in the current window. Note that a bit-map can be used to indicate only current window expiries. Traversal of 64 elements takes constant time. If the far future expiries have a tendency to be deleted too often, then the cost of sorted insertion gets wasted.

Inactive expiries are stored in *MC* array and then moved to *IPW* array just as in case of hierarchical timing wheel. Inactive expiries are kept unsorted. Therefore *HandleOutWindowExpiries* still consumes O(n) time if all the expiries form part of inactive expiries. *StartTimer* primitive takes O(n) time to insert the expiry node in the respective slot's linked list. It has been observed from experiments that bit-map calendar queue has a nearly constant execution time for *HandleOutWindowExpiries* if the partition has no inactive expiries.

This algorithm consumes more memory than all other algorithms described before. It requires 64 pointers and 10+*p* memory to store inactive expiries for every partition. Therefore each partition requires O(N+*C*) space where N is the total number of processes in the respective partition and *C* the constant number of bytes for bit-maps and arrays. The code complexity and therefore the execution time of this algorithm are much higher than other algorithms because the traversal of slot's linked list involves the check for major cycle and partition window. *HandleTimeOutExpiry* while handling closer expiries now needs to check if each request is intended to expire in the current window.

5.5.4 Summary

As we see from the above algorithms, insertion in one-shot timer approach requires O(n) time. It has a poor algorithmic performance primarily due to the fact that the algorithms try to reduce the timer-reprogramming overhead. The primary reason for maintaining the near future expiries in a sorted way is to reduce the timer-reprogramming overhead. The timer

may have to be programmed in all the primitives. Thus a poor timer-reprogramming strategy can increase the time complexity of these primitives.

It becomes essential that both in-window and out-window expiries must be handled to have a fair distribution of time complexity. If out-window expiries are ignored in *StartTimer* primitive, then *HandleOutWindowExpiries* will have the burden of handling it. If *HandleOutWindowExpiries* loops through all processes (or PCBs), then having O(1) or O(logn) insertion algorithm becomes more beneficial than O(n) insertion since it reduces the worst-case time complexity of the primitive. A $O(n^2)$ algorithm for *HandleOutWindowExpiries* becomes too costly as the application is deprived of executing to its full window duration.

Advantages

1. The one-shot timer approach has better accuracy needed for real-time systems.
2. It avoids the unnecessary cache pollution like that in tick-based approach.
3. It provides a low process release delay.

Disadvantages

1. One-shot timer approach has the timer-reprogramming overhead.
2. One-shot timer approach has a poor algorithmic performance.
3. It is difficult to estimate the WCET of the primitives and hence causes problem in estimating the WCET of PES and critical sections. This further creates problems for verification.

5.6 Firm Timers

Firm timers use soft timers in combination with one-shot timer wherein the one-shot timer is programmed to expire after an overshoot time. Soft timers avoid interrupts by checking for expired timers at strategic points in the kernel such as system call, interrupt and exception return paths. These checks are called as soft timer checks. In soft timer concept, the kernel completely relies on soft timer checks to detect expiries. This is clearly not a suitable approach for real-time systems since the delay is unbounded. Whereas one-shot timer still has the problem of interrupt handling overhead and cache pollution to a considerable extent. In some real-time systems it might be required to eliminate even such interrupts. As the frequency of timer events increases, the interrupt-handling overhead grows.

Interrupts are asynchronous events that cause an uncontrolled context switch and result in cache pollution. Cache pollution affects the worst-case execution time (WCET) predictability of real-time applications and pose problems for verification of software. At strategic points,

the working set in the cache is likely to be replaced anyway and hence polling and dispatching timers does not cause significant additional overhead. In essence, soft timers allow voluntary switching of context at convenient moments.

Firm timers increase the overhead of polling within kernel. Firm timers can introduce timer latency if these soft timer checks occur infrequently or the distribution of the checks and the timer deadlines are not well matched. It has been observed in [18] that firm timers are effective if the ratio on the number of soft timers that fire to the number of soft timer checks is sufficiently large. The overshoot parameter can be configured to suit a particular partition's requirement. If the overshoot parameter is made zero, then it turns out to be a one-shot timer approach and if the overshoot parameter is made very large, then it becomes soft timers. As soon as the soft timer check detects an expiry, the one shot timer is reprogrammed to expire at the next expiry, thus avoiding an interrupt at the current expiry. Figure 5.11 shows how firm timers work.

In our implementation, soft timer checks are added at the beginning of all system calls (APEX service), which will increase the probability of finding an expiry at the earliest. The execution control will be with the application until the system call is called and as soon as the control enters the kernel, the first thing it would do is to check for an expiry. At this stage the working set of the cache would be replaced with a new set. Soft timer check has also been added in idle process that runs when no processes are ready to run. Our experimental results show that the soft timer check in idle process results in significant firing of soft timers with low overhead.

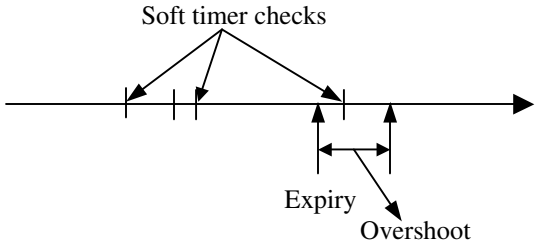


Figure 5.11: Firm timers

Since the soft timer checks are inserted in the kernel code, it becomes very essential that a careful analysis be done to find the proper location of the check and the check itself should be of low overhead. These checks increase the code size and also consume some time to check. Hence the number of instructions in this check is kept at minimum. The checks themselves should be kept at a considerable distance in time so as to have a uniform distribution of checks. On proper analysis, the firm timers can be tailored to reduce the critical section overhead. If the overshoot value is larger than the longest critical section, then

we can have a soft timer check immediately at the exit of critical section or at safe points within the critical section. This can eliminate the need of entry and exit critical section. In a normal case, firm timers can increase the code size and execution time of the service call due to soft timer checks.

Firm timers can perform poor if the application is more computational intensive and rarely invoke system calls. This makes it dependent on the behavior of the application. The worst-case process release delay in case of firm timer will now have the overshoot parameter in it.

$$\textit{Worst-case Process Release Delay}^{\textit{Firm-timer}} = \textit{Overshoot time} + \textit{Process Release Delay}$$

5.7 Summary

Designing an algorithm for partitioned system has many challenges as highlighted in this chapter. In hierarchical timing wheel, obtaining the hierarchical information for an out window expiry involves considerable effort. This chapter discusses on how the hierarchical timing wheel can be used for partitioned system. The lack of efficient one-shot timer algorithms has been addressed in this chapter by using bit-maps in case of hierarchical multiple linked list and bit-map calendar queue algorithms. One must note the fact that all the algorithms discussed in this chapter functions on the basis of strict time partitioning.

This chapter emphasizes the fact that timing wheel algorithms have constant insertion time complexity and is the most suitable algorithm for real-time systems under tick-based approach. In the same way, using bit-maps for one-shot timer algorithms promises better hopes in improving one-shot timer algorithm performance for real-time systems.

Chapter 6

Experimental Results

This chapter studies tick-based, one-shot and firm timer approaches and assess their suitability for partition-based system. It also analyses the performance of algorithms. The experiments use rate monotonic setup derived from Hartstone benchmark [30] that is used to assess the performance of hard real-time systems. Appendix B gives more details about this benchmark.

The experiments are carried out on OASYS 653 running on PowerPC MPC 565 processor used widely in the avionics industry. A simulated ARINC 653 based avionics application is used to stress the time management module of OASYS 653. The PowerPC processor (MPC 565) on which the experiments were carried out doesn't have a cache and hence parameters related to cache is not measured. The data has been collected using visionCLICK debugger tools of Wind River systems.

Since the system already has strict time partitioning, the timing of events in one partition will not affect other partitions. All partitions in our system are protected in time. Since the partitions execute in a strict time partitioning environment, the timing behavior of individual partitions or applications are independent. The algorithms handle the time-out requests of the partitions independently and therefore the timing events and the way these timing events are handled by the algorithms for a given partition is in no way related to another partition. Hence a single partition is considered for the experiments.

6.1 Performance of Approaches

The experimental setup has a total of five partitions and has the schedule shown in figure 6.1. This schedule is derived from ARINC 653 standard shown in figure 1.1 and hence is more realistic in nature. Partition **A** is the partition of our interest. All other partitions are in IDLE state and do not interfere in any way with partition **A**. The duration of partition **A** is 20ms

and its periodicity is 100ms. The length of major cycle is 200ms. With this experimental setup we measure the performance of timer approaches.

Baseline setup: In the baseline setup, partition **A** has a total of 15 processes, 8 periodic and 7 aperiodic processes. Generally the periodicity of a process is a multiple of the periodicity of the respective partition. The priority of a periodic process is rate-monotonic in nature. Higher the number, higher is its priority. The absolute deadline of periodic processes is equal to its period. The periodicities of the periodic processes are harmonic. The absolute deadline of aperiodic process is kept infinite for convenience. All these process parameters are set before the system starts running and hence the system is completely static.

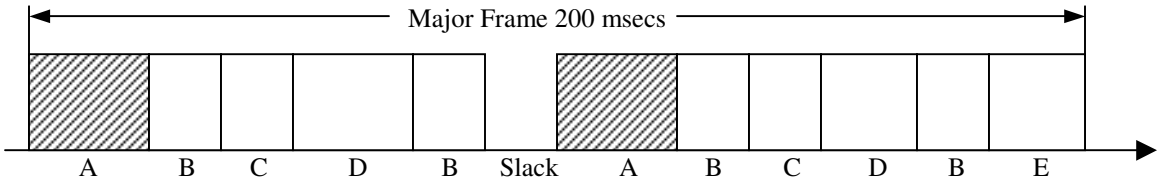


Figure 6.1: Partition schedule of experimental setup

The experiments of hartstone benchmark test the system than the time management and hence cannot be used directly. The processes call APEX services with time-out request. The whole experimental setup is designed in such a way to ensure that there are sufficient calls to all the time management primitives and all the major conditions get covered. This is to have a fair measure of time management performance. In this experiment, 30 major cycles have been chosen as the duration to collect significant amount of data.

The periodicity and priority of the processes are given in table 6.1. Process P0 is the main controlling process that acquires a semaphore on which other periodic processes wait with a time-out. P0 after acquiring the semaphore goes for a wait and hence all other periodic processes expire, failing to acquire the resource. The time-out requests has an expiry distribution of Poisson with $\lambda = 10$. The Poisson distribution is formed by a program that generates numbers (refer Appendix C), which are used as difference between successive expiries on a time scale. These numbers are stored in a table and are used to test all the three time management approaches. This is to ensure that the approaches are tested under the same conditions. In order to avoid identical application environment in every partition window, the application requests time-outs with varying samples of Poisson distribution input that expire with different offsets in every partition window. Only processes from P1 to AP3 request time-outs that expire with Poisson distribution property. Process AP6 and AP7 expire in the inactive window of the partition for every 2.5 major cycle.

Name	Nature	Priority	Periodicity (ms)
P0	Periodic	63	100
P1	Periodic	58	100
P2	Periodic	58	100
P3	Periodic	56	200
P4	Periodic	56	200
P5	Periodic	53	400
P6	Periodic	53	400
P7	Periodic	51	800
AP1	Aperiodic	50	-
AP2	Aperiodic	50	-
AP3	Aperiodic	50	-
AP4	Aperiodic	5	-
AP5	Aperiodic	5	-
AP6	Aperiodic	2	-
AP7	Aperiodic	2	-

Table 6.1: Experimental Setup

This experimental setup has 100% processor utilization for partition **A**. There is no scope to execute idle process. The low priority processes AP4 and AP5 initialize a matrix and perform matrix multiplication all the time. The APEX services are called to ensure that timer interrupts are generated inside critical sections. In this experimental setup, n = 13.

6.1.1 Tick-based

Process release delay: The priority of the process doesn't affect the process release delay. The data has been collected for different tick widths to indicate the affect of tick frequency on process release delay. Table 6.2 shows the process release delay with timing wheel algorithm. The data is collected for processes P1, P5 and AP2, of which P1 is a high priority process, P5 is a medium priority process and AP2 is a low priority process. These processes are selected at random and their process release delay indicates primarily the affect of tick frequency. The release delay may also comprise the affect of critical section delay but this has been found to be negligible.

The data given in table 6.2 ignores the process release delay caused due to PERIODIC_WAIT service because all the periodic processes gets released at the start of partition window and do not have tick width delay. The data has been collected only for processes that expire within a partition window (and therefore gets affected by tick frequency) and the expiry forms part of Poisson distribution.

Tick width (µs)	P1 (µs)		P5 (µs)		AP2 (µs)	
	Average	Worst	Average	Worst	Average	Worst
100	68.64	104	57.3	85	30.64	55
200	141.35	203	104.76	210	99.59	155
300	118.66	217	221.76	301	134.42	224
400	352.27	396	108.07	324	171.35	398
500	424.5	499	169	234	200.7	324

Table 6.2: Process release delay of Tick-based Approach

From our experiments we have observed that with tick width of 50µs, there were 7 tick interrupts that were generated inside critical sections. We observed that the worst-case delay faced due to critical section was 13µs. From table 6.2, we can infer that the process release delay increases as tick width increases and tick width forms the major part of process release delay. The worst-case process release delay is directly proportional to the tick width. Table 6.2 reveals that if a process has a sensitive deadline, then it has a higher tendency to miss its deadline in tick-based approach.

The expiry distribution affects the process release delay in case of tick-based approach. For example, if the distribution is Poisson with $\lambda = 10$, then in a small gap of time (say in a tick width) while at the peak of the curve, there are more number of expiries and hence the timer queue length (or the length of linked list) for that tick width is large. This concept is expressed in figure 6.2.

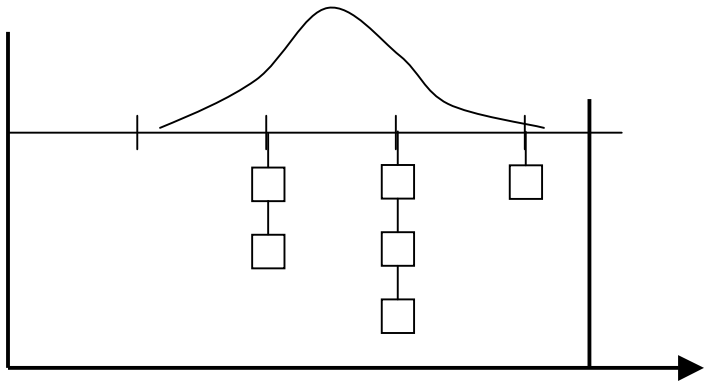


Figure 6.2: Expiry distribution vs. timer queue length

Process release delay forms a major delay component for any high priority process to be released. That is, if the priority of the process released is higher than the priority of the

process currently executing, then process release delay becomes an important factor. However, a queue length does affect on how early a high priority process can start executing. A high priority process being at the head of queue is released quicker but may have to wait till all low priority processes down the queue gets released.

Table 6.3 shows the relationship between the tick width and timer queue length. As the tick width increases, the average length of timer queue also increases and hence affects the process release delay and the scheduling latency for a high priority process. The insertion time of timing wheel algorithm is constant and hence is not affected by the timer queue length.

Tick width (µs)	**Average queue length**	**Worst queue length**
100	1.135	3
200	1.265	4
300	1.335	4
400	1.395	4
500	1.515	5

Table 6.3: Tick frequency vs. Timer queue length

Interrupt Overhead: The number of tick interrupts generated in a partition window is inversely proportional to the tick width. Let N_h be the total number of tick interrupts generated by partition **A** for 30 major cycles. Let C_h be the cost of every tick interrupt. It has been found that each interrupt has an interrupt latency of 11µs and therefore C_h = 11µs. Table 6.4 lists the number of tick interrupts and the percentage of interrupt overhead under each tick width.

Tick width	**N_h**	**% of interrupt overhead**
100	11861	10.87
200	5940	5.445
300	3960	3.63
400	2940	2.695
500	2340	2.145

Table 6.4: Interrupt overhead

Our observations from experiments reveal that tick frequency has least effect on the total time consumed by the primitives for a given duration. The total time consumed by time management primitives is basically driven by the performance of the algorithm chosen, especially for *StartTimer* and *HandleOutWindowExpiries*. Hence these parameters are discussed in later sections.

The choice of fixing a better tick frequency (for an application/partition) should primarily be on the permissible process release delay due to tick frequency and the interrupt overhead for that tick frequency.

6.1.2 One-shot timer

Process release delay: Dual Linked Lists (DLL) is used to measure the process release delay of one-shot timer. The choice of the algorithm doesn't make any difference to the process release delay because only the primitive *HandleTimeOutExpiry* affects the process release delay and this primitive has similar behavior for all the algorithms. Table 6.5 lists the process release delays for processes P1, P5 and AP2. One-shot timer has a better process release delay than tick-based approach. Here the process release delay includes the ISR prologue (6µs) and the primitive latency.

We have found from our experiments that there were around 5 interrupts generated inside critical section and the worst case critical section delay faced is around 22µs. The worst-case critical section delay of one-shot timer is poorer than tick-based approach (13µs). The *StartTimer* primitive called inside most critical sections take O(n) time in DLL as compared to timing wheel which take O(1) time. This clearly stresses the necessity of O(1) insertion algorithm for one-shot timer in order to have a lower WCET of critical section.

Process	**Average process release delay (µs)**	**Worst process release delay (µs)**
P1	12.38	14
P5	11.61	14
AP2	13.01	24

Table 6.5: Process release delay of One-shot timer approach

Interrupt Overhead: Partition **A** in the experimental setup generates a total of 422 interrupts for 30 major cycles ignoring the partition switch interrupt. Therefore $N_h = 422$ and $C_h = 11\mu s$. Therefore the total interrupt overhead of this approach for 30 major cycles is

4642µs, which amounts to 0.386% of the total time allocated for partition **A** in 30 major cycles.

6.1.3 Firm timers

We have carried out different types of experiment with the same experimental setup, to assess the performance of firm timers. The soft timer check has just 1µs of overhead.

6.1.3.1 Experiment – 1

In this experiment a soft timer check is added at the entry of every service call. A service call is an entry point to the kernel where we hope to capture more number of expiries at the earliest. Partition **A** has 100% utilization of the processor with no scope to execute idle process.

Process Release Delay: Table 6.6 lists the process release delays of P1, P5 and AP2 for different overshoots.

Overshoot	P1 (µs)		P5 (µs)		AP2 (µs)	
	Average	Worst	Average	Worst	Average	Worst
0	12.6	14	16.83	26	12.63	18
5	15.75	18	14	30	16.15	20
10	18.75	25	20.16	35	19.93	24
50	28.32	63	29.08	62	42.15	63
100	12.37	113	67.58	113	72.12	113
200	19.13	213	129	190	112.22	213
300	19.13	312	164.16	312	164.03	313

Table 6.6: Process release delay of Firm timer approach

Table 6.6 reveals the fact that the worst-case process release delay faced by a process is proportional to the overshoot value. The soft timer checks of low priority processes fires expiries of high priority processes. From average process release delay of P1 we observe that P1 being the high priority process faces the least amount of delay even for large overshoots because on an average P1 finds that its expiries are more often captured by soft timer checks. AP2 being a low priority aperiodic process, its expiries can be captured by soft timer checks of lower priority processes only and since there are less number of processes that has priority

lesser than the priority of AP2, it faces a higher average process release delay. Therefore an expiry of high priority process has more chances of getting captured by soft timer checks if it has more number of low priority processes that are ready to run, provided the low priority processes yield to kernel points.

Most often the hard timer is hit for small overshoots. As the value of overshoot increases, the number of expiries captured by the soft timer check increases. When a high priority process gets released with a delay due to large overshoots, it gives an opportunity to execute low priority processes and their soft checks, until the high priority process gets released. The probability that a soft timer check releasing a process increases as the overshoot increases.

Overhead: Let N_h indicate the number of hard timer interrupts generated or hard timers fired, N_s the number of soft timers fired and N_c the number of soft timer checks. Let C_h be the cost of firing a hard timer, C_s the cost of firing a soft timer and C_c the cost of soft timer check. In our case C_h = 11µs (interrupt latency), C_s = 3µs and C_c = 1µs. The total number of timers that must fire in a given interval of time is $N_t = N_h + N_s$. Table 6.7 lists the number of soft timers and hard timers fired for different values of overshoot.

Overshoot	**N_s**	**N_h**	**N_c**
0	0	422	61198
5	63	359	61891
10	172	283	61681
50	239	138	61638
100	241	138	62363
200	271	66	60024
300	258	63	62176

Table 6.7: Soft timers and Hard timers fired

From table 6.7, we can infer that as overshoot increases, the number of expiries captured by soft timer check also increases and the number of hard timer interrupts reduces. It has been mentioned in [18] that firm timers are effective if the ratio on the number of soft timers fired to the number of soft timer checks is sufficiently large, i.e $N_s/N_c > C_c/(C_h - C_s)$. This is true if $C_cN_c + C_hN_h + C_sN_s < C_hN_t$. For this experiment this is not true because N_c, the number of soft timer checks is large. Hence, firm timers are not effective for the experiment under consideration.

Considering the worst-case situation for an overshoot value of 100µs, we have an overhead of, 62122 (62363 – 241) x 1µs + 241 x 3µs + 138 x 11µs = 64353µs. Therefore this

experiment has achieved process release delays of table 6.6 with an overhead of around 5.36% for partition **A** in 30 major cycles.

6.1.3.2 Experiment – 2

In experiment-1, there were no scopes to run idle process because the aperiodic processes AP4 and AP5 consumed all the idle time available and their priority was higher than idle process. Experiment-2 tries to reduce the soft timer overhead by introducing idle process, which has a soft timer check inside the idle loop. In this experiment, soft timer checks inside service calls are removed; instead a soft timer check is added in the process scheduler and the idle process. Process scheduler is the place where the context switch happens and this is an ideal place to have a soft timer check.

In experiment-2, the aperiodic processes AP4 and AP5 run for 4 major cycles and suspend themselves for 2 major cycles that gives an opportunity to run idle process. Here the CPU utilization is not 100%. The soft timer check inside the idle process does not contribute to any overhead, as the check is being done during idle time. Table 6.8 lists the process release delays for this experiment.

The overhead comprises only the soft timer check inside process scheduler. For an overshoot value of 100µs, ignoring the idle process soft timer check, N_c = 1086, N_h = 225 and N_s = 122. Therefore overhead = 3805µs, which is just 0.317% for partition **A** in 30 major cycles. Hence, table 6.8 reveals that process release delays can be improved significantly with the introduction of soft timer check in idle process with just 0.317% of overhead. Our experiments reveal that idle process soft timer check can release a process as early as 4µs which proves that firm timers can outperform one-shot timer if an application entertains idle process and this can be achieved with lesser overhead.

Overshoot	**P1 (µs)**		**P5 (µs)**		**AP2 (µs)**	
	Average	**Worst**	**Average**	**Worst**	**Average**	**Worst**
5	13.64	20	17.08	26	13.89	20
10	17.32	24	21.66	32	17.62	24
50	46.7	64	38.91	63	37.2	63
100	85.73	114	55.5	112	64.51	114

Table 6.8: Process release delay for experiment-2

6.1.3.3 Experiment – 3

An advantage of firm timers is that it enables to set different overshoot values to different processes. A more practical way to use firm timers is to set an overshoot value that is inversely proportional to a process's priority. That is, a high priority process needs to be released earliest and cannot tolerate any amount of delay. Such processes can be set with lower values of overshoot. A low priority process can be set with higher values of overshoot. Thus as the priority of a process changes in dynamic priority scheduling, the overshoot of that process also changes. Our experimental setup is based on fixed priority scheduling and hence the overshoot value for a process is fixed. A high priority process is more likely to be released by hard timers since it has low overshoot value.

Table 6.9 lists the process release delays for the same setup as experiment-1 but there are less number of soft timer checks inserted. Note that this setup has 100% processor utilization and therefore there is no idle process soft timer check. In this experiment, soft timer checks are added in process scheduler and in services that do not pass time-out requests. This is because the services through which time-outs are requested will also call process scheduler and for which the soft timer check in process scheduler is sufficient.

In this experiment, N_c = 2065, N_h = 343 and N_s = 37. Therefore overhead = 5912µs which is 0.492% for partition **A** in 30 major cycles. These numbers clearly indicate that the number of soft timer checks have been reduced and therefore the number of soft timers fired (N_s) also have been reduced. This means that most of the processes are released due to hard timers that fire after overshoot amount of time. Hence the processes have a poor process release delay and the firm timer is not effective in this experiment.

Process	Priority	Overshoot	Average	Worst
P1	58	5	16.77	20
P5	53	50	45.16	63
AP2	50	150	117.74	164

Table 6.9: Process release delay for experiment – 3

The performance of firm timers depends on how often application yields control to service calls or kernel. In order to get better benefits from firm timers with low overhead, the soft timer check must be inserted at various points within the kernel by studying the application's behavior. It is important that firm timers have low overhead for partitioned system since already the time management consumes significant amount of time to manage time-outs.

6.1.4 Comparisons

1. *Process release delay*: The process release delay of tick-based approach is primarily driven by the tick frequency. The process release delay of one-shot timer is driven by expiry distribution and hence is dependent on the application behavior. The process release delay of firm timer is primarily driven by the overshoot value. In case the application needs a low process release delay, then one-shot timer approach is the most suitable approach for hard real-time applications.
2. *Overhead*: Tick based approach has a higher overhead than one-shot timer and firm timers. Firm timers have the next higher overhead. On a system that has cache, firm timers can help in reducing the cache pollution and its overhead.
3. *Flexibility*: Both tick-based approach and firm timer approach provides the flexibility of configuring the tick frequency and the overshoot respectively to suit an application's need. This in turn can help in controlling the process release delay and the overhead of the approach. In a partitioned system, different applications run with different criticality levels and both tick-based and firm timers provide the flexibility to the applications.

6.2 Algorithmic Performance

The aim of the following experiments is to measure the performance of primitives, *StartTimer* and *HandleOutWindowExpiries* of various algorithms discussed in chapter 5. The performance is measured for different values of n, which is the total number of time-out requests per partition. It is known from chapter 5, that both the primitives *StopTimer* and *HandleTimeOutExpiry* have O(1) performance, hence is independent of n. Therefore the performances of these primitives are out of the scope of interest.

The experimental setup is designed to have one inactive expiry for every three active window expiries. The experiments are designed keeping the conscious timing model as the base wherein low priority processes that execute with low frequency request time-outs that expire farther down the time line and vice-versa. The experimental setup has near future, future and far future expiries. The experimental setup is based on rate-monotonic setup (just as in case of Hartstone benchmark) and the aperiodic processes have infinite deadline. All the processes are actively involved in time-out requests, wherein each process requests for a current window expiry and an out window expiry. Apart from this the processes are not involved in any other functionalities as these actions do not affect the time management performance in any way and therefore the experimental setup is kept simple. The setup do not invoke *StopTimer* primitive and hence all the requests expire with certainty.

The expiries are generated with a realistic approach based on Poisson distribution. Only the current window expiries form part of this distribution and the out window expiries are generated by programming the values. The application or the partition under consideration generates time-out requests that expire with $\lambda = 5$ for the first 10 major cycles. This is considered as the initialization phase of the application wherein we can expect modest timer expiries. After initialization the partition enters a stable mode, wherein time-outs expire with $\lambda = 10$. Once in every 20 major cycles, we may expect a heavy duty of timer expiries (in case of an event or error) and this is simulated with $\lambda = 1$, that generates very closer expiries. This is the default setup used in the experiments for current window expiries. The algorithms are compared with n = 10, 30 and 70, wherein n is the number of processes or time-out requests in a partition.

6.2.1 Tick-based algorithms

Timing Wheel (TWL) and Hierarchical Timing Wheel (HTWL) algorithms are compared for n = 10 and 30. Since these algorithms have constant time performance, they are not analyzed for large values of n.

For n = 10: The experimental setup has 10 periodic processes with periodicity 100ms, 200ms, 400ms and 800ms. The experimental setup has the partition schedule shown in figure 6.3. The same setup is also used to compare one-shot timer algorithms under n = 10. Partition **B** is the partition of interest for n = 10. Each window of Partition **B** has duration of 10ms. The partition has a periodicity of 100ms. Note that it has two partition windows in each of its period, separated by 40ms. In a given major cycle, partition **B** has 4 active windows and 5 inactive windows. Therefore p = 4. The partition schedule of figure 6.3 is derived from ARINC 653 and hence is a realistic example.

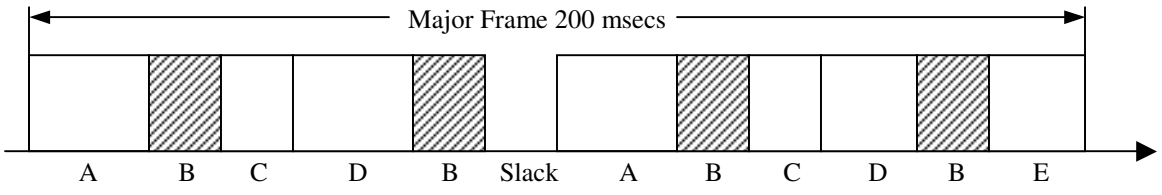


Figure 6.3: Partition schedule

Table 6.10 lists the performance of tick-based algorithms. The data has been collected by running the system for 60 major cycles. All measurements are in microsecond. ACET represents Average-Case Execution Time and WCET represents Worst-Case Execution

Time. The tick width is set to 200µs, which is comparable with the slot width under the same experimental setup for one-shot timer.

n = 10	StartTimer		HandleOutWindowExpiries	
	ACET	WCET	ACET	WCET
TWL	1.51	4	40.4	54
HTWL	7.74	15	35.57	56

Table 6.10: Algorithmic performance of tick-based algorithms for $n = 10$

Observations:

1. In case of *StartTimer* primitive, hierarchical timing wheel has a higher execution time than timing wheel due to the complexity involved to find and maintain the hierarchical information.
2. *HandleOutWindowExpiries*:
   a. The performance of this primitive depends on inactive expiries and current window expiries. More the number of these expiries, higher the execution time of this primitive. TWL and HTWL have O(1) insertion and hence the primitive is not affected by the expiry distribution or the queue length.
   b. Although hierarchical timing wheel is expected to perform better for out window expiries, its complexity to manage the hierarchy slightly outweighs its advantage for small n. Hierarchical timing wheel still has a better average performance than timing wheel for handling out window expiries.

For $n = 30$: The partition schedule considered for $n = 30$, is the same as shown in figure 6.1. Larger partition window duration (20ms) is chosen since the window needs to accommodate more number of expiries ($n = 30$). Hence partition **A** of figure 6.1 is chosen for $n = 30$. It has a periodicity of 100 ms. It has two active windows in every major cycle and hence $p = 2$. The partition has 18 periodic processes and 12 aperiodic processes. The periodic processes have periodicity of 100ms, 200ms, 300ms, 600ms and 1200ms. The tick width is set to 300µs.

With the use of DELAYED_START service, periodic processes can be released with a delay from the start of partition window. This is called as delayed release. This avoids the problem of critical instant that occurs if all the processes are released simultaneously and the highest priority process faces release delay due to low priority processes. Experiments have shown that the release pattern, such as simultaneous release (in case of critical instant) or

delayed release of processes can affect the performance of *HandleOutWindowExpiries* primitive. This is shown in table 6.11.

n = 30	StartTimer		HandleOutWindowExpiries			
			Delayed release		Simultaneous release	
	ACET	WCET	ACET	WCET	ACET	WCET
TWL	1.76	4	110.3	145	117.8	178
HTWL	7.23	15	93.05	127	103.57	160

Table 6.11: Algorithmic performance of tick-based algorithms for n = 30

Observations:

1. By comparing the *StartTimer* primitive of table 6.11 with table 6.10, it is clear that tick-based algorithms are not affected by *n*, the input and hence runs in constant time. This makes the algorithm ideal for real-time systems. This is true because insertion time complexity of the algorithms are independent of *n*.
2. In case of *HandleOutWindowExpiries*, hierarchical timing wheel performs better than timing wheel, since the latter traverses the entire list of processes and the former scans only the required processes, to handle out window expiries. Therefore finding and maintaining hierarchical information has proven its merit for large values of *n* and hence is better than timing wheel.
3. As expected the WCET of *HandleOutWindowExpiries* is higher for simultaneous release because the primitive has to release more number of processes at the start of partition window inside partition event server. In case of delayed release, some of the processes will be released later.

Time consumed by time management: For *n* = 30 and for 60 major cycles, it has been found that timing wheel algorithm altogether consumes 1.58% of total time allocated for partition **A** (execution time of all the primitives included). Similarly hierarchical timing wheel consumes 2.38% of time, which is larger as the algorithm consumes more time to determine and maintain the hierarchy.

6.2.2 One-shot timer algorithms

Four algorithms, Dual-Linked Lists (DLL), Multiple Linked Lists (MLL), Hierarchical Multiple Linked Lists (HMLL) and Bit-map Calendar Queue (BCQ) are compared for *n* =

10, 30 and 70. The experiments are carried out with delayed release pattern for processes. The experimental setup has Poisson distribution for current window expiries. Table 6.12 denotes the time complexity notation of these algorithms for the primitives.

Algorithms	StartTimer		HandleOutWindowExpiries	
	Worst	Expected	Worst	Expected
DLL	O(n)	θ(n)	O(n)	θ(n)
MLL	O(n)	θ(1)	$O(n^2)$	θ(n)
HMLL	O(n)	θ(1)	$O(n^2)$	θ(n)
BCQ	O(n)	θ(1)	O(n)	θ(n)

Table 6.12: Time complexity notations of one-shot timer algorithms

For *n* = 10: The experimental setup is the same as used in tick-based algorithm for *n* = 10. The partition schedule considered here is also the same as shown in figure 6.3. Table 6.13 lists the performance of one-shot timer algorithms. The data has been collected for 60 major cycles and the measurements are in microsecond.

n = 10	StartTimer		HandleOutWindowExpiries	
	ACET	WCET	ACET	WCET
DLL	5.57	10	27.74	43
MLL	2.33	9	44.25	61
HMLL	7.55	14	35.39	57
BCQ	8.30	14	56.89	78

Table 6.13: Algorithmic performance of one-shot timer algorithms for n = 10

Observations:

1. *StartTimer*:
    a. MLL has a better insertion time complexity than other algorithms. This is due to the fact that the linked list is broken into multiple linked lists, reducing the average length of a slot's queue and therefore the insertion time. MLL ignores out window expiries, whereas HMLL doesn't. Therefore HMLL has a higher execution time than MLL. The WCET of HMLL and BCQ also involve the time taken to insert out window expiries.

b. The performance of DLL basically depends on the time-out request pattern. The experiment is designed to have typical time-out request pattern mentioned in section 4.5.2. Insertion to primary linked list of DLL exploits this property for current window expiries.
c. BCQ has a higher ACET than HMLL as the former inserts the future active window expiries into the corresponding slot's linked list in a sorted way, whereas HMLL inserts future expiries in the hierarchical array in O(1) time (unsorted way).

2. *HandleOutWindowExpiries*:
   a. BCQ has a larger WCET than other algorithms due to the code complexity involved to handle inactive expiries. Primarily the benefit of BCQ can be seen only if the partition has lesser number of inactive expiries. Overall, BCQ is not a good choice for small n, due to its code complexity.
   b. DLL inserts future expiries in secondary linked list as part of *StartTimer* primitive. Hence *HandleOutWindowExpiries* requires only adjusting the time-out value relative to the current window properties, apart from handling inactive expiries. Therefore this algorithm having the least amount of work runs faster.
   c. MLL handles out window expiries in a naïve way just like timing wheel and hence has a larger execution time than HMLL, whereas HMLL handles future expiries in an intelligent way.

For n = 30: The experimental setup and partition schedule is the same as used for tick-based algorithms in the case of n = 30. Table 6.14 lists the performance of one-shot timer algorithms for n = 30 along with the affect of expiry distribution. The experimental setup considered for n = 30 has only one form of Poisson distribution (with $\lambda = 10$) and Bi-modal distribution. Poisson and Bi-modal distribution samples are generated using programs that are based on Poisson and Bi-modal equations respectively. Appendix C lists the programs used to generate these distributions.

HandleOutWindowExpiries is responsible for releasing the processes for the first time. The first releases of the processes have random distribution. The experiment encourages only subsequent time-out requests of the processes to have Poisson or Bi-modal distribution. Therefore Poisson or Bi-modal distribution does not affect *HandleOutWindowExpiries*. If the first release of the processes has any of these distribution properties, then the primitive is affected by the expiry distribution. The affect of this distribution is visible only in *StartTimer* primitive for the experimental setup under consideration.

n = 30	StartTimer				HandleOutWindowExpiries	
	Poisson		Bi-modal			
	ACET	WCET	ACET	WCET	ACET	WCET
DLL	8.069	23	7.94	23	70.72	99
MLL	2.46	7	2.67	10	127.47	183
HMLL	7.025	14	7.2	14	96.84	145
BCQ	8.69	16	8.94	16	94.65	130

Table 6.14: Algorithmic performance of one-shot timer algorithms for n = 30

Observations:

1. *StartTimer*: The affect of distribution can be seen clearly in case of MLL. The WCET in case of Poisson ($\lambda = 10$) is less than Bi-modal distribution. This indicates that, Bi-modal distribution has larger slot queue length than Poisson distribution ($\lambda = 10$). DLL is not affected by the expiry distribution. The WCET of HMLL and BCQ also involve inserting future expiries and hence the affect of expiry distribution alone cannot be captured.
2. *HandleOutWindowExpiries*: A noted observation from table 6.14 is that, BCQ has a lower WCET compared to that of table 6.13. The benefit of BCQ is now becoming evident for larger values of *n* (surpassing its code complexity), since the algorithm has already inserted active window expiries in *StartTimer* primitive.

For *n* = 70: The partition schedule considered for $n = 70$ is same as that of figure 6.1, except that partition **D** is the partition of our interest. Partition **D** has window duration of 30ms, enough to accommodate the timer requests of 70 processes. It has two active windows in every major cycle and hence $p = 2$. It has a periodicity of 100ms. The partition has 35 periodic processes and 35 aperiodic processes. The periodic processes have periodicity of 100ms, 400ms, 800ms and 1200ms. Table 6.15 lists the performance of one-shot timer algorithms for $n = 70$. The data has been collected by running the system for 100 major cycles and the measurements are in microsecond.

Figure 6.4 and 6.5 summarizes the primitive performance (WCET) of one-shot timer algorithms for different values of *n*. The current window expiry distribution has Poisson property.

n = 70	StartTimer		HandleOutWindowExpiries	
	ACET	WCET	ACET	WCET
DLL	11.62	48	133.38	188
MLL	2.75	11	233.96	405
HMLL	7.73	14	157.16	291
BCQ	10.65	27	140.63	248

Table 6.15: Algorithmic performance of one-shot timer algorithms for n = 70

Figure 6.4: StartTimer

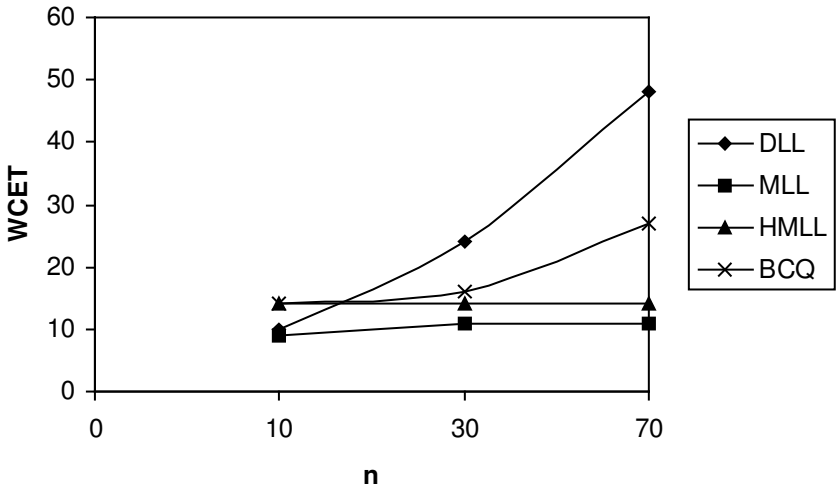
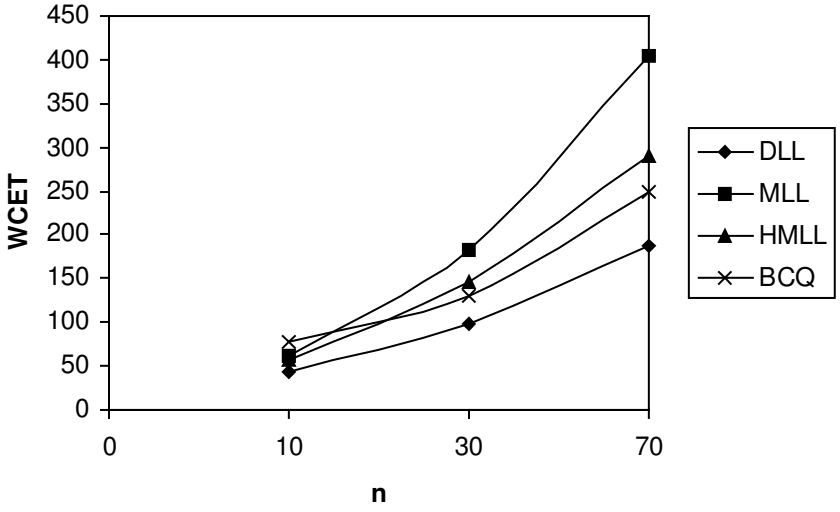


Figure 6.5: HandleOutWindowExpiries



6.2.3 Summary

1. *Tick-based approach*: Timing wheel algorithm is a better algorithm for tick-based approach, which is very suitable for hard real-time systems, but it fails to have stable property. Hierarchical timing wheel has better WCET for *HandleOutWindowExpiries* than timing wheel and can be considered as the next option, if the execution time of this primitive is considered critical. It also has stable property. Tick-based algorithms are better for estimating the WCET of critical sections and kernel services given their constant time performance. However, this has been achieved with loss of accuracy and larger interrupt overhead.
2. *One-shot timer*: The one-shot timer algorithms offer different algorithmic options out of which only few qualify to be used in hard real-time systems. These options have been achieved with determinism and high precision accuracy of one-shot timer and with lower interrupt overhead. However, the algorithms also have larger memory consumption.
   - From figure 6.4, it is clear that as *n* increases, WCET of *StartTimer* in case of DLL also increases rapidly making it dependent on the input, *n* (as cited in [37]). This makes it inappropriate for real-time systems. But given its simplicity and superior performance for *HandleOutWindowExpiries*, it could still be considered for small *n*. DLL does not get affected by expiry distribution.
   - MLL offers the best WCET performance for *StartTimer* among all other algorithms, however it performs very badly for *HandleOutWindowExpiries*. Expiry distribution also affects MLL and has the least memory consumption compared to other MLL-based algorithms. MLL does not have stable property just like timing wheel.
   - HMLL is the most suitable one-shot timer algorithm for hard real-time systems, considering its stable performance for all values of *n*. The experiments show that to a large extent, *StartTimer* primitive performance is independent of *n*. It also has a reasonable *HandleOutWindowExpiries* performance and memory consumption.
   - BCQ also performs reasonably well for all values of *n* and also has good *HandleOutWindowExpiries* performance. However it may perform badly in presence of larger inactive expiries due to its code complexity. Therefore among all the algorithms considered, BCQ gets affected by the timing model of the application and may perform poorly for non-conscious timing model. It still has poor performance for small values of *n*, which can be a common case in partitioned systems. The memory consumption of BCQ is larger than all other algorithms.

In case of HMLL, the WCET of *StartTimer* primitive has shown consistent performance of 14µs for all values of *n*. This is because as we see from section 5.5.2.2 the primitive has two

parts for servicing every request. The first part will handle current window expiries by inserting the request into the slot's linked list with O(n) complexity. In the second part, if the request is not a current window expiry, then it would be a out window expiry which gets inserted into the respective array (*MC*, *PW* or *IPW*) with O(log*p*) complexity. Clearly the first part of the primitive has the same time complexity as that of MLL's *StartTimer* primitive, whose WCET is no more than 11µs (i.e. in the case of $n = 70$). This means that WCET of the primitive in case of HMLL has come from the second part of the primitive which has O(log*p*) complexity. We know that the experimental setup in case of $n = 10$ has $p = 4$ and the worst-case binary search comparisons for this would be 3. In case of $n = 30$ and 70, $p = 2$ and the worst-case binary search comparisons here would be 2. But from the fact that WCET of the primitive is 14µs for both $n = 30$ and 70, makes us clear that the worst-case total comparisons of binary search for $n = 10$ also has been 2. Therefore the WCET of *StartTimer* primitive in case of HMLL for the experimental setup under consideration is governed by the second part of the primitive that handles out window expiries, which has a constant time performance.

Extending the argument for all scenarios, we may conclude that skewed distribution of current window expiries, just like MLL, raises the WCET of HMLL *StartTimer* primitive. As we observed from our results above, both the experimental setups of $n = 10$ and 70 had skewed distribution of $\lambda = 1$, and the WCET of the primitive was no worse than 11µs.

In case of BCQ, the algorithm also inserts out window active expiries into the corresponding slot's linked list, which can be fluctuating as both current window, and out window expiries coincide within a slot. Therefore we observe varying values of WCET for BCQ. Note that the slot width is 157µs (10ms / 64) in case of $n = 10$, 313µs (20ms / 64) in case of $n = 30$ and 469µs (30ms / 64) in case of $n = 70$. The slot width also has influence on the length of the linked list and therefore the WCET of the primitive. Dividing the partition window duration by 64 may not give us a remainder of zero, which means that the slot width is a rounded up value. For example, slot width = 20ms / 64 = 312.5 = 313µs.

Hence, we may conclude that HMLL presented in this thesis has an overall superior performance than other algorithms and is the most suitable one-shot timer algorithm for real-time systems.

Time consumed by time management: Following data includes the total execution time of all the primitives collected for $n = 30$. It indicates the percentage of time consumed by time management primitives for partition **A** over 60 major cycles. BCQ and HMLL having higher code complexity have the larger execution time.

- DLL – 2.87%, MLL – 2.405%, HMLL – 2.98%, BCQ – 3.36%

Chapter 7

Conclusions and Future Work

From all the approaches discussed in this thesis we understand that each approach has its own advantages and disadvantages. An intelligent judgment to choose the best approach and the best algorithm requires a careful assessment of several factors.

A tick-based approach has constant time algorithms. Thus the WCET of critical sections and thereby OS services can be predicted for verification of the software. One can provide the option of configuring the tick frequency to applications. But the tick-based approach is based on frequent interrupts, which not only pollutes the cache but also has a larger overhead. Tick-based approach also has poor process release delay.

One-shot timer approach has high accuracy and generates interrupts only when required. It also has a better way of handling closer expiries. It has a better worst-case process release delay achieved with low overhead. But one-shot timer approach is believed to have poor algorithmic performance. Some of the new algorithms presented in this thesis show some promise towards improving the performance.

Firm timers can provide better average process release delays than one-shot timer. But this comes at a cost of soft timer checks and its associated overhead. Experiments reveal that one can benefit from firm timer if more number of soft timer checks result in capturing the expiries. This approach is application dependent. Experiments also reveal that soft timer checking inside idle process can improve the process release delay with negligible overhead. Overshoot can also be made as a configurable parameter.

Experiments also reveal that HMLL is the most suitable one-shot timer algorithm for real-time systems and for majority values of n.

For the future work, the benefit of firm timers in reducing the affect of cache pollution needs to be explored. We also need to study on how firm timers can be used effectively to reduce the critical section overhead within kernel.

The performance of bit-map based multiple linked lists has to be thoroughly understood for different expiry distributions, varying slot widths and large ranges of n. This can be further analyzed for different bit-map representations such as 32 and 128. The performance can be compared against other algorithms for non-partitioned systems as well. Other algorithms such as priority queue and median pointer linked list can also be implemented and their performance measured for partitioned system.

Bibliography

[1] Federal Aviation Administration and NASA. Partitioning in Avionics Architectures: Requirements, Mechanisms and Assurance. Final report. March 2000.

[2] Lei Luo and Ming-Yuan Zhu. Partitioning Based Operating System: A Formal Model. *ACM SIGOPS Operating Systems Review*, Volume 37, Issue 3, July 2003.

[3] Airlines Electronic Engineering Committee. ARINC 653-1, Avionics Application Software Standard Interface. October 2003.

[4] John Rushby. Bus Architectures For Safety-Critical Embedded Systems. Lecture Notes In *Computer Science*; Vol. 2211. *Proceedings of the First International Workshop on Embedded Software*. 2001

[5] T. Carpenter et al. ARINC 659 Scheduling: Problem Definition. *Proceedings in 1994 Real-time Systems Symposium*. December 1994.

[6] Federal Aviation Administration. Commercial Off-The-Shelf Real-Time Operating System. Final report. February 2004.

[7] T.P.Baker and Alan Shaw. The Cyclic Executive Model and Ada. *Real-Time Systems*. June 1989.

[8] Neil Audsley and Andy Wellings. Analysing APEX Applications. *Proceedings of the 17th IEEE Real-Time Systems Symposium*. 1996.

[9] Daeyoung Kim. Strongly Partitioned System Architecture For Integration of Real-Time Applications. *PhD Thesis*. University of Florida. 2001.

[10] Darren Cofer and Mural Rangarajan. Formal verification of overhead accounting in an avionics RTOS. *Proceedings of the 23rd IEEE Real-Time Systems Symposium*. 2002.

[11] Z. Deng, J.W.S Liu. Scheduling Real-Time Applications in an Open Environment. *Proceedings of the 18th IEEE Real-Time Systems Symposium*. 1997.

[12] Iain John Bate. Scheduling and timing analysis for safety-critical real-time systems. *PhD Thesis*. Department of Computer Science. University of York. 1998.

[13] Jane W.S.Liu. Real-Time Systems. Prentice Hall Publication, 2000.

[14] C.L.Liu and J.W.Leyland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *Journal of the ACM 20(1)*. January 1973.

[15] C Douglass Locke, David R. Vogel and Thomas J. Mesler. Building a Predictable Avionics Platform in Ada: A Case Study. *Proceedings of the IEEE Real-Time Systems Symposium*. December 1991.

[16] George Varghese and Tony Lauck. Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility. *Proceedings of the eleventh ACM Symposium on Operating systems principles*. February 1996.

[17] Randy Brown. Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM*. Volume 31, Issue 10 October 1988.

[18] Ashvin Goel. Operating system support for low latency streaming. *PhD Thesis*. IIT Kanpur. July 2003.

[19] Ashvin Goel et al. Supporting Time-Sensitive Applications on a Commodity OS. *ACM SIGOPS Operating Systems Review*. Volume 36. 2002.

[20] Paul Parkinson. Safety-Critical Software development for Integrated Modular avionics. WindRiver.

[21] Mohit Aron and Peter Druschel. Soft-timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*. Volume 18. August 2000.

[22] Clark Williams. Linux Scheduler Latency. Red Hat Inc. March 2002.

[23] Peter Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proceedings of 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. January 2003.

[24] Rick Siow Mong Goh and Ian Li-Jin Thng. MLIST: An Efficient Pending Event Set Structure for Discrete Event Simulation. *International Journal of Simulation*. Volume 4. December 2003.

[25] Adam M. Costello and George Varghese. Redesigning the BSD Callout and Timer Facilities. *Software-Practice & Experience*. Volume 28, Issue 8. July 1998.

[26] T.Cormen, C.Leiserson and R.Rivest. Introduction to Algorithms. MIT Press/McGraw Hill. 1990.

[27] Michael Goodrich and Roberto Tamassia. Algorithm Design. John Wiley and Sons. Inc.

[28] Kah Leong Tan and Li-Jin Thng. SNOOPY Calendar Queue. In *Proceeding of the 2000 winter Simulation Conference*.

[29] Dan Tsafrir, Yaov Etsion and Dror G. Feitelson. General-Purpose Timing: The Failure of Periodic Timers. Technical Report.

[30] Patrick Donohoe, Ruth Shapiro and Nelson Weiderman. Hartstone Benchmark User's Guide. Version 1.0. March 1990.

[31] Jean J. Labrosse. MicroC/OS – II The Real-Time Kernel. R&D Books, Miller Freeman publication. 1999.

[32] Christo Angelov and Jesper Berthing. A Jitter-Free Kernel for Hard Real-Time Systems. Lecture notes in Computer Science.

[33] Robert Ronngren and Rassul Ayani. A Comparative Study of Parallel and Sequential Priority Queue Algorithms. *ACM Transactions on Modeling and Computer Simulation*. Volume 7, Issue 2. April 1997.

[34] MPC565/MPC566 User's Manual.

[35] Alan Grigg and Neil C. Audsley. Towards a Scheduling and Timing Analysis solution for Integrated Modular Avionic systems. Microprocessors and Microsystems Journal. 1999.

[36] http://www.oberle.org/apic_timer-timers.htm

[37] C.M.Reeves. Complexity Analyses of Event Set Algorithms. Computer Journal. Volume 27, Issue 1. February 1984.

[38] J.H.Blackstone, G.L.Hogg and Don T. Philips. A two-list synchronization procedure for discrete event simulation. *Communications of the ACM*. Volume 24. December 1981.

[39] Paul Wyman. Improved event scanning mechanisms for discrete event simulation. *Communications of the ACM*. Volume 18. June 1975.

[40] H. Kopetz. The Time-Triggered Model of Computation. *Real-Time Systems Symposium*. The 19th IEEE Volume. December 1998.

Appendix A

The major features of MPC 565, a member of the Motorola MPC500 RISC microcontroller family are as follows:

- An MPC500 core with floating point unit
- 36 Kbytes of static RAM
- 1 Mbyte of flash memory
- Unified System Integration Unit (USIU), a flexible memory controller and improved interrupt controller
- Three time processor units
- A 22-timer channel modular I/O system
- JTAG and background debug mode (BDM)

The RISC processor (RCPU) used in the MPC500 family of microcontrollers integrates five independent execution units: an integer unit (IU), a load/store unit (LSU), a branch processing unit (BPU), a floating-point unit (FPU) and an integer multiplier divider (IMD). The RISC's use of simple instructions with rapid execution times yields high efficiency and throughput for MPC500-based systems. Most integer instructions execute in one clock cycle. Instructions can complete out of order for increased performance; however, the processor makes execution appear sequential.

Major features of the RCPU include:

- High-performance microprocessor
  — Single clock-cycle execution for many instructions
- Five independent execution units and two register files
- Facilities for enhanced system performance
  — Atomic memory references
- In-system testability and debugging features
- High instruction and data throughput
  — Branch-folding capability during execution (zero-cycle branch execution time)
  — Programmable static branch prediction on unresolved conditional branches
  — A pre-fetch queue that can hold up to four instructions
  — Interlocked pipelines with feed-forwarding that control data dependencies in hardware

Figure A shows the modules on MPC565.

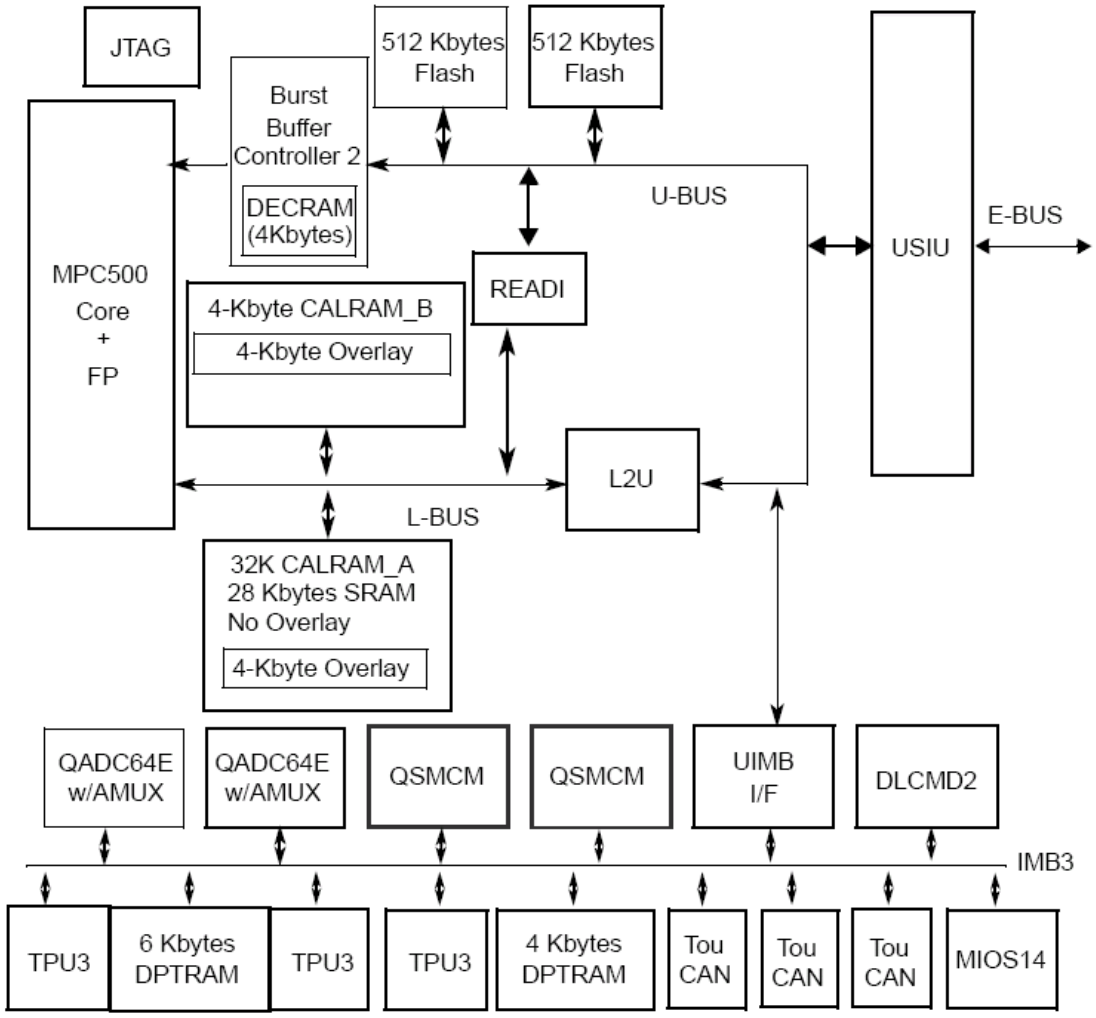


Figure A: MPC 565 Block Diagram

Time Base (TB)

The Time Base (TB) is a 64-bit free running binary counter defined by the MPC565 architecture. The TB has two independent reference registers which can generate a maskable interrupt when the time base counter reaches the value programmed in one of the two reference registers. The period of the TB depends on the driving frequency. The TB is clocked by the TMBCLK clock. The 32-bit decrementer timer is also driven by the same clock source. If the bit TBS of System Clock Control Register (SCCR) is set to 0, then TMBCLK is divided by 4, which selects the resolution to 1µs. The state of TB is not affected

by any resets and should be initialized by software. Reads and writes to the TB are restricted to special instructions.

Two reference registers are associated with the time base: TBREF0 and TBREF1. A maskable interrupt is generated when the TB count reaches to the value programmed in one of the two reference registers. Two status bits in the time base control and status register (TBSCR) indicate which one of the two reference registers generated the interrupt.

Appendix B

The Hartstone benchmark comprises a series of requirements to be used for testing the ability of a system to handle hard real-time applications. Its name is derived from Hard Real Time and the fact that the computational workload of the benchmark is provided by a variant of the Whetstone benchmark. "Hard" real-time applications *must* meet their deadlines to satisfy system requirements; this contrasts with "soft" real-time applications where a statistical distribution of response time is acceptable. The rationale and operational concept of the Hartstone benchmark describes five test series of increasing complexity and one of these, the Periodic Harmonic (PH) Test Series, is described in detail.

Periodic Harmonic (PH) Test Series

The Periodic Harmonic (PH) Test Series is the simplest of the five test series defined for the Hartstone benchmark. It consists of a set of five periodic Ada tasks that are independent in the sense that their execution need not be synchronized; they do not communicate with each other. Each periodic task has a *frequency*, a *workload*, and a *priority*. Task frequencies are harmonic: the frequency of a task is an integral multiple of the frequency of any lower-frequency task. Frequencies are expressed in Hertz; the reciprocal of the frequency is a task's period, in seconds. A task workload is a fixed amount of work, which must be completed within a task's period. The *workload rate*, or speed, of a task is equal to its per-period workload multiplied by the task's frequency. The *deadline* for completion of the workload is the next scheduled activation time of the task. Successful completion on time is defined as a *met deadline*. Failure to complete the workload on time results in a *missed deadline* for the task. Missing a deadline in a hard real-time application is normally considered a system failure. In the Hartstone benchmark, however, processing continues in order to gather additional information about the nature of the failure and the behavior of the benchmark after deadlines have begun to be missed. Therefore, in the Ada implementation of the PH series, if a task misses a deadline it attempts to compensate by not doing any more work until the start of a new period. This process, called *load-shedding*, means that if a deadline is missed by a large amount (more than one period, say) several work assignments may be cancelled. Deadlines ignored during load-shedding are known as *skipped deadlines*. The reason for

load-shedding is that "resetting" offending tasks and letting the test series continue allows more useful information to be gathered about the failure pattern of the task set.
Task priorities are assigned to tasks according to *rate-monotonic* scheduling discipline. This means that higher-frequency tasks are assigned a higher priority than lower frequency tasks. The priorities are fixed and distinct. The rate-monotonic priority assignment is optimal in the sense that no other fixed-priority assignment scheme can schedule a task set that cannot be scheduled by the rate-monotonic scheme. In the Hartstone task set, priorities are statically assigned at compile time via the Priority pragma. Task 1 has the lowest priority and task 5 has the highest. The main program, which starts these tasks, is assigned a priority higher than any task so that it can activate all tasks via an Ada rendezvous.

Hartstone Experiments: Four *experiments* have been defined for the PH series, each consisting of a number of *tests*. However these experiments cannot be used for testing time management module of an ARINC 653 kernel since Hartstone benchmark is not designed for partitioned system nor does it comply with ARINC 653 standard. The Hartstone benchmark doesn't test time management module of the system on which it is used and it only tests the systems ability to handle hard real-time applications. Our experiments stress the time management module in particular.

Appendix C

The following program is used to generate Poisson distribution for $\lambda = 10$.

```
unsigned int Poisson(unsigned int seed)
{
        double sum; long int ran = 0; long int k;

        sum = 0;
        do{
                ran++;
                sum = sum + expon(10, seed); /* exponential function */
        }while(sum < 1.0);
        ran--;
        return ran;
}

double expon(double b, int sd)
{
        double u;

        u = randam(sd);
        if(u < 0.0001)
             u = 0.0001;
        u = -log(u)/b;
        return u;
}

double randam(long int sd)
{
        const long int C = 25173;
        const long int D = 13849;
        const long int M = 32768;
```

```
        double val;

        sd = (C*sd + D);
        val = sd % M;
        val = val/32768.0;

        return val;
}
```

The Bi-modal distribution is generated by using the following program.

```
unsigned int Bi_modal(unsigned int seed)
{
        double val, val1;

        val = unif(0.0, 1.0, seed); /*uniform distribution */
        if(val < 0.1)
            val1 = 9.5238;
        else
            val1 = 0.0;
        val = 0.95238*val +  val1;
        return (unsigned int)(100 *val);
}

double unif(double a, double b, long int seed)
{
        return (b-a)*randam(seed) + a;
}
```

The values generated by these programs are stored in a table and are used by processes to request time-out using TIMED_WAIT APEX service that expire with the desired distribution property. The data values obtained from these programs are scaled to fit within a partition window.

Appendix D

OASYS – 653

Embedded systems for safety critical applications often integrate multiple functions and must generally be fault tolerant. In hard real time applications, it must be guaranteed by design that all real-time transactions will produce the correct result and meet their deadlines. These requirements lead to a need for mechanisms and services that provide protection against fault propagation and ease the construction of distributed fault tolerant applications. A key feature of an avionics RTOS is its ability to meet tight processing deadlines. An application program controlling the release of a weapon may require an action from the operating system in less than one-thousandth of a second. The avionics computer resource is an embedded generic computing platform that is able to host multiple applications with different levels of criticality. Safety-critical applications such as flight control, cockpit display, navigation, radar control etc., run on an avionics computer resource. These applications need strong assurance from the operating system that their hard real-time requirements are met. More importantly, they require the assurance that in presence of faults, a fault in one application should not propagate to the others.

Partitions

Automated aircraft control has traditionally been divided into distinct functions that are implemented separately (e.g., autopilot, auto-throttle, flight management); each function has its own fault tolerant computer system. A by-product of this federated architecture is that faults are strongly contained within the computer system of the function where they occur and cannot readily propagate to affect the operation of other functions. The obvious disadvantage to the federated approach is its profligate use of resources: each function needs its own computer system (which is generally replicated for fault tolerance), with all the attendant costs of acquisition, space, power, weight, cooling, installation and maintenance. Integrated Modular Avionics (IMA) has therefore emerged as a design concept with a single computer system (with internal replication to provide fault tolerance) provides a common computing resource to several functions. As a shared resource, IMA has the potential to diminish fault containment between functions, so any realization of IMA must provide

partitioning to ensure that the shared computer system provides protection against fault propagation from one function to another. Partitioning uses appropriate hardware and software mechanisms to restore strong fault containment to such integrated architectures.

The advantages of having partitions are:

1. Allow applications with different criticalities to co-exist and run on the same core module without causing any potential damages to other partitions/applications.
2. Allow integrating and reusing applications written on a federated architecture.
3. Provide the flexibility to add enhancement to an application without modifying the schedule or any other applications.
4. Making modifications in one partition and not have to 'regression-test' the others because partition protection is in place.

ARINC 653

ARINC 653 is a standard specification for avionics application software drafted by Airlines Electronic Engineering Committee in 1997. ARINC stands for Aeronautical Radio INC. ARINC 653 standard is a general purpose APEX (APplication EXecutive) interface for an avionics computer's operating system and the application software. The standard includes interface requirements as well as list of services that allow the application software to control the scheduling, communication and status information of its internal processing elements.

Central to the ARINC 653 philosophy is the concept of partitioning, whereby functions resident on a core module are partitioned with respect to space (memory partitioning) and time (temporal partitioning). A partition is therefore a program unit of the application designed to satisfy these partitioning constraints. The OS is responsible for enforcing partitioning and managing individual partitions. A robustly partitioned system allows partitions with different criticality levels to execute in the same core module, without affecting one another spatially or temporally. The time partitioning ensures that activities in one partition do not disturb the timing of events in other partitions. The space partitioning ensures that software in one partition cannot change the software or private data of another partition either in memory or in transit.

OASYS - 653

OASYS-653 is an indigenously developed ARINC 653 compliant real-time operating system that hosts multiple applications with different levels of criticality on a single processor. It supports all the APEX services of ARINC 653-1. OASYS-653 kernel provides bounded

execution time with deterministic features. OASYS-653 provides strict time partitioning and ensures deterministically that no application overruns.

All the memory components required by the kernel and the partition is statically allocated from configuration table during system startup time. Therefore the kernel eliminates memory fragmentation. All the hardware resources are protected with access permissions. The health monitoring (HM) function of the OS is responsible for monitoring and reporting hardware, application and OS software faults and failures. The HM helps to isolate faults and prevent failures from propagating. All other modules of ARINC 653 such as interpartition and intrapartition communication are supported by default in OASYS-653.

OASYS-653 has been tested with ARINC 653-3 conformity test specification. Currently ARINC 653-2 extended services is being implemented. OASYS-653 has Ethernet driver and UDP/IP stack on PowerPC. Currently OASYS-653 runs on PowerPC and ARM processors.

Hardware Interface Layer

HIL is a lower level layer that is closely dependent on the hardware. It provides services for context switch; interrupt control, timer control and others. This layer enables the operating system to be independent of the hardware. The services provided by HIL are called from the kernel to control and monitor the hardware devices.

Strict Time Partitioning

OASYS-653 provides guaranteed timing window for each application. It supports strict time partitioning wherein under no circumstance an application is allowed to overrun. It ensures that no partition violates partitioning principles. It has deterministic one-shot timer facilities and a very reliable time management module.

ARINC 653-1 compliant APEX interface

The APEX (APplication EXecutive) provides a common logical environment for the application software. This environment enables independently produced applications to execute together on the same hardware.

Constant time algorithms

The kernel uses constant time algorithms that ease the software verification process. Hence it deterministically provides bounded execution time for all system calls. It doesn't support dynamic memory allocation.

ARINC 653 Partition Scheduler

ARINC 653 defines a two level scheduling model, wherein the partitions are scheduled based on cyclic execution model and processes within a partition are scheduled based on priority-based preemptive algorithm.

Deterministic Process Scheduler

In a traditional priority scheduler, the time taken to insert a process in the priority ordered ready queue takes O(n) time where 'n' is the number of processes. This implementation has an indeterministic behavior wherein the time taken by the scheduler depends on the input.

Our kernel implements a deterministic process-scheduling algorithm using bit-maps. Here the time taken to perform ready queue operations take constant time. The scheduler supports 64 priority levels.

No priority inversion

The kernel doesn't use semaphores to protect its critical section, instead uses simple flags. Hence this keeps the kernel lighter and avoids the priority inversion problem.

OASYS-653 Certification package

Accord provides complete (DO-178B) certification life-cycle package for all safety-critical products. Accord supports its customers throughout the certification process by customizing the following items to suit avionics application development.

- Plans
  - Plan for Software Aspects for Certification (PSAC)
  - Software Development Plan (SDP)
  - Software verification plan (SVP)
  - Software configuration management plan (SCMP)
  - Software quality assurance plan (SQAP)

- Standards
  - Requirements Standards
  - Design Standards
  - Coding Standards
- Software requirements specification
- Software design
  - Architectural design
  - Detailed design
- Source code
- Software Review records
  - Requirements
  - Design
  - Coding
- Software verification test cases and results
- Problem reports
- Traceability matrices
- Software quality assurance records
- Software Accomplishment Summary (SAS)