Design Patterns For Scheduling Tasks In Real-Time Systems

U.V.R. Sarma[1], Dr. K. V. Chalapati Rao[2] and Dr. P. Premchand[3]
[1]CVR College of Engineering, Department of CSE, Ibrahimpatan, R.R.District, A.P., India
Email: sarmauvr@yahoo.co.in
[2]CVR College of Engineering, Department of CSE, Ibrahimpatan, R.R.District, A.P., India
Email: chalapatiraokv@gmail.com
[3]Osmania University, Department of CSE, Hyderabad, A.P., India
Email: profpremchand.p@gmail.com

Abstract—**This paper discusses some of the popular design patterns employed in scheduling tasks in various categories of real-time applications and generalizes the guidelines for designing and developing real-time systems. A tool is proposed to automate the application of these patterns when creating a detailed design model.**

Index Terms—**Design Pattern, Real-Time Systems, periodic, aperiodic and sporadic tasks, EDF, SETF, LETF, Rate-Monotonic Scheduling, Deadline-Monotonic Scheduling, Priority Ceiling, Priority inversion, Least Laxity, Maximum Urgency First.**

I. INTRODUCTION

A *design pattern* is a generalized solution to a commonly occurring problem [1]. It is not a finished design that can be transformed directly into code – rather, it is a description or template, which helps in solving a problem and can be used in many different situations. Good OO designs are reusable, extensible and maintainable. Patterns show us how to build systems with good OO design qualities. Gamma and others, popularly known as Gang of Four (GoF) list 23 design patterns [1]. Usage of design patterns helps to lower software costs.

A system is said to be *real-time* when quantitative expressions of time are necessary to describe the behavior of the system. Usually, there are many real-time tasks in such a system and are associated with some time constraints. A real-time task is classified into either hard, firm or soft real-time depending on the consequences of a task failing to meet its timing constraints. A real-time task is called *hard* if missing its deadline may cause catastrophic consequences on the environment under control. It is called *firm* if missing its deadline makes the result useless, but missing does not cause serious damage and it is called *soft* if meeting its deadline is desirable (e.g. for performance reasons) but missing does not cause serious damage.

Typical Hard RT activities include sensory data acquisition, detection of critical conditions and low-level control of critical system components. Typical application areas are power-train control, air-bag control, steer by wire, brake by wire (automotive domain) and engine control, aerodynamic control (aircraft domain). Typical Firm RT Activities include decision support and value prediction. Typical application areas are Weather forecast, Decisions on stock exchange orders etc. Typical Soft RT Activities include command interpreter of user interface, keyboard handling, displaying messages on screen, representing system state variables and transmitting streaming data. Typical application areas are communication systems (voice over IP), user interaction and comfort electronics (body electronics in cars).

Appropriate *scheduling* of tasks is the basic mechanism adopted by a real-time operating system to meet the constraints of a task. Therefore, selection of an appropriate task scheduling algorithm is central to the proper functioning of a real-time system.

II. TYPES OF REAL-TIME TASKS

A task is an executable entity of work, characterized by task execution properties, task release constraints, task completion constraints and task dependencies. Tasks are executed in a system made up of a processor (CPU) and other resources (communication links, shared data etc.)

Task execution properties include Worst-case execution time, Criticality level, Preemptive / non-preemptive execution and whether a task can be suspended or not during execution.

Based on the way real-time tasks recur over a period of time, it is possible to classify them into three main categories: *periodic, aperiodic* and *sporadic* tasks.

A periodic task is one that repeats after a certain fixed time interval. Deadlines have to be met with precision, and so they are hard real-time in nature, whereas an aperiodic task can arise at random instants but we can afford to miss few deadlines and hence soft real-time in nature. A sporadic task is one that recurs at random instants and mostly hard real-time in nature.

III. SCHEDULING APPROACHES

Three commonly used approaches to scheduling Real-Time Systems are [2]:

1. Clock-Driven approach.
2. Round-Robin approach
3. Priority-Driven approach.

3.1 Clock-Driven approach (also called time-driven)

In this approach, decisions on task execution are made at specific time instants. All the parameters of the tasks are fixed and known apriority. Schedules are computed off-line and stored for use at run-time. Hence scheduler overhead during run-time is minimized. H/W timer is used to regularly space time instants. Scheduler selects the task, blocks, and after expiry of timer, awakes and repeats these actions.

3.2 Round-Robin approach

This approach follows the fairness principle – justice to all. It is commonly used for scheduling time-shared applications (time slice – in the order of tens of msec). A FIFO queue of tasks ready for execution is maintained. An executing task is preempted at the end of time slice. 1/nth share for n tasks in a round (hence called processor-sharing algorithm). A variation to this approach is weighted round robin. Different tasks may have different weights – for ex., a task with weight wt gets wt time slices every round; the length of the round is equal to the sum of the weights of all the ready tasks; weights can be adjusted to speed up / retard the progress of each task.

Round Robin approach is not suitable for tasks requiring good response time, particularly when we have precedence constrained tasks, but suitable for incremental consumption – for ex., UNIX pipe. In case of pipelining, they can complete even earlier – ex: Transmission of messages by switches en route in a pipeline fashion. The approach does not require a sorted priority queue; it uses only a round-robin queue; which is a distinct advantage, since priority queues are expensive.

3.3 Priority-Driven approach

These algorithms never leave any resource idle intentionally. Scheduling decisions are made when events such as releases and completion of tasks occur; hence called event-driven. Other names – greedy scheduling, list scheduling and work-conserving scheduling.

Most scheduling algorithms used in non-real-time systems are priority-driven. Examples include FIFO, LIFO – priorities based on release times; SETF (Shortest-Execution-Time-First) and LETF (Longest-Execution-Time-First) – priorities based on execution times. Priority-driven algorithms can be implemented with either preemptive or non-preemptive scheduling. In some cases, non-preemptive scheduling may look attractive, but, in general, non-preemptive scheduling is not better than preemptive scheduling.

Priority-based scheduling systems operate in one of three primary modes:

1. Static priority systems,
2. Semi-static or
3. Dynamic priority systems.

3.3.1 Static Priority System: In a static system, a task's priority is determined at compile time and is not changed during execution. This has the advantages of simplicity of implementation and simplicity of analysis. The most common way of selecting task priority is based on the period of the task, or, for asynchronous event-driven tasks, the minimum arrival time between initiating events. This is called *Rate Monotonic Scheduling* (RMS). Static scheduling systems may be analyzed for schedulability using mathematical techniques such as *Rate Monotonic Analysis*.

Another well-known fixed priority algorithm is the *Deadline-Monotonic* (DM) algorithm. This algorithm assigns priorities to tasks according to their relative deadlines: the shorter the relative deadline, the higher the priority. Clearly, when the relative deadline of every task is proportional to its period, the RMS and DM algorithms are identical. When the relative deadlines are arbitrary, the DM algorithm performs better in the sense that it can sometimes produce a flexible schedule when the RMS algorithm fails, while the RMS algorithm always fails when the DM algorithm fails.

3.3.2 Semi-Static Priority System: Semi-static priority systems assign a task a nominal priority but adjust the priority based on the desire to limit priority inversion. This is the essence of the priority ceiling pattern. If a low priority task locks a resource needed by a high priority task, the high priority task must block itself and allow the low priority task to execute, at least long enough to release the needed resource. The execution of a low priority task when a higher priority task is ready to run is called priority inversion. The naïve implementation of semaphores and monitors allows the low priority task to be interrupted by higher priority tasks that do not need the resource. Because this preemption can occur arbitrarily deep, the priority inversion is said to be unbounded. It is impossible to avoid at least one level of priority inversion in multitasking systems that must share resources, but one would like to at least bound the level of inversion. This problem is addressed by the priority ceiling pattern. The basic idea of the priority ceiling pattern is that each resource has an attribute called its priority ceiling. The value of this attribute is the highest priority of any task that could ever use that particular resource. The active objects have two related attributes: nominal priority and current priority. The nominal priority is the normal executing priority of the task. The object's current priority is changed to the priority ceiling of a resource it has currently locked as long as the latter is higher.

3.3.3 Dynamic Priority System: Dynamic priority systems assign task priority at run-time based on one of several possible strategies. The three most common dynamic priority strategies are:

1. Earliest Deadline First
2. Least Laxity
3. Maximum Urgency First

In Earliest Deadline First (EDF) scheduling, tasks are selected for execution based on which has the closest deadline. This algorithm is said to be *dynamic* because task scheduling cannot be determined at design time, but only when the system runs. In this algorithm, a set of tasks is schedulable if the sum of the task loadings is less than 100%. This algorithm is optimal in the sense that if it is schedulable by other algorithms, then it is also schedulable by EDF. However, EDF is not stable; if the total task load rises above 100%, then at least one task will miss its deadline, and it is not possible to predict in general which task will fail. This algorithm requires additional run-time overhead because the scheduler must check all waiting tasks for their next deadline frequently. In addition, there are no formal methods to prove schedulability before the system is implemented.

Laxity for a task is defined as the time to deadline minus the task execution time remaining. Clearly, a task with a negative laxity has already missed its deadline. The algorithm schedules tasks in ascending order of their laxity. The difficulty is that during run-time, the system must know expected execution time and also track total time a task has been executing in order to compute its laxity. While this is not conceptually difficult, it means that designers and implementers must identify the deadlines and execution times for the tasks and update the information for the scheduler every time they modify the system. In a system with hard and soft deadlines, the Least Laxity (LL) algorithm must be merged with another so that hard deadlines can be met at the expense of tasks that must meet average response time requirements (see MUF, below). LL has the same disadvantages as the EDF algorithm: it is not stable; it adds run-time overhead over what is required for static scheduling, and schedulability of tasks cannot be proven formally.

Maximum Urgency First (MUF) scheduling is a hybrid of RMS and LL. Tasks are initially ordered by period, as in RMS. An additional binary task parameter, criticality, is added. The first n tasks of high criticality that load under 100% become the critical task set. It is this set to which the Least Laxity Scheduling is applied. Only if no critical tasks are waiting to run are tasks from the noncritical task set scheduled. Because MUF has a critical set based on RMS, it can be structured so that no critical tasks will fail to meet their deadlines.

IV. Patterns

With this background of various scheduling approaches in Real-Time Systems, we can now suggest suitable patterns known as Execution Control Patterns for scheduling tasks. They deal with the policy by which tasks are executed in a multitasking system. This is normally executed by the Real-Time Operating System, if present. Most Real-Time Operating Systems offer a variety of scheduling options. The most important of these are listed here as execution control patterns [3].

1. Cyclic Executive Pattern for simple task scheduling.
2. Time Slicing – Round Robin Pattern – fairness in task scheduling.
3. Static Priority Pattern – preemptive multitasking for schedulable systems.
4. Semi-static priority – Priority Ceiling Pattern.
5. Dynamic Priority Pattern – preemptive multitasking for complex systems.

The primary difference occurs in the policy used for the selection of the currently executing task.

4.1 Cyclic Executive Pattern

In the cyclic executive pattern the kernel (commonly called the *executive* in this case) executes the tasks in a prescribed sequence. Cyclic executives have the advantage that they are “brain-dead” simple to implement and are particularly effective for simple repetitive tasking problems. Also, it can be written to run in highly memory-constrained systems where a full RTOS may not be an option. However, they are not efficient for systems that must react to asynchronous events and not optimal in their use of time. There have been well-publicized cases of systems that could not be scheduled with a cyclic executive but were successfully scheduled using preemptive scheduling. Another disadvantage of the cyclic executive pattern is that any change in the executive time of any task usually requires a substantial tuning effort to optimize the timeliness of responses. Furthermore, if the system slips its schedule, there is no guarantee or control over which task will miss its deadline preferentially.

Figure 1.[4] shows how simple this pattern is. The set of threads is maintained as an ordered list (indicated by the constraint on the association end attached to the *Abstract Thread* class). The Cyclic Executive merely executes the threads in turn and then restarts at the beginning when done. When the Scheduler starts, it must instantiate *all* the tasks before cycling through them.

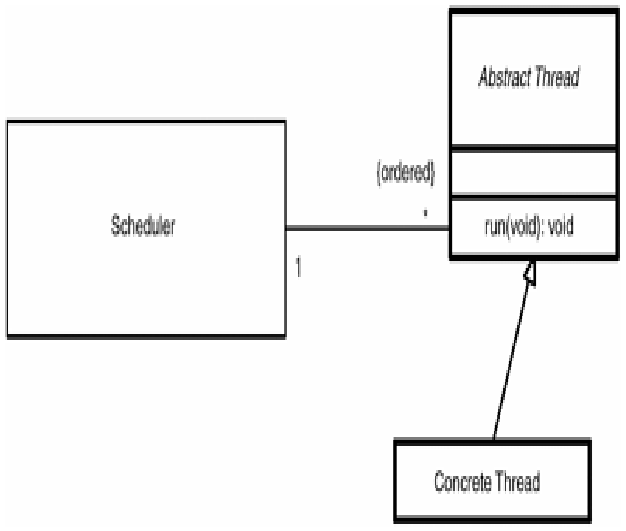

Figure 1.Cyclic Executive Pattern

4.2 Time Slicing – Round Robin Pattern

The kernel in the time slicing pattern executes each task in a round-robin fashion, giving each task a specific period of time in which to run. When the task’s time budget for the cycle is exhausted, the task is preempted and the next task in the queue is started. Time slicing has the same advantages of the cyclic executive but is more time based. Thus it becomes simpler to ensure that periodic tasks are handled in a timely fashion. However, this pattern also suffers from similar problems as the cyclic executive. Additionally, the time slicing pattern doesn’t “scale up” to large numbers of tasks well because the slice for each task becomes proportionally smaller as tasks are added.

The Round Robin Pattern is a simple variation of the Cyclic Executive Pattern. The difference is that the *Scheduler* has the ability to preempt running tasks and does so when it receives a tick message from its associated *Timer.* Two forms of the Round Robin Pattern are shown below. The complete form (Figure 2a [4]) shows the infrastructure classes *Task Control Block* and *Stack.* The simplified form (Figure 2b [4]) omits these classes.

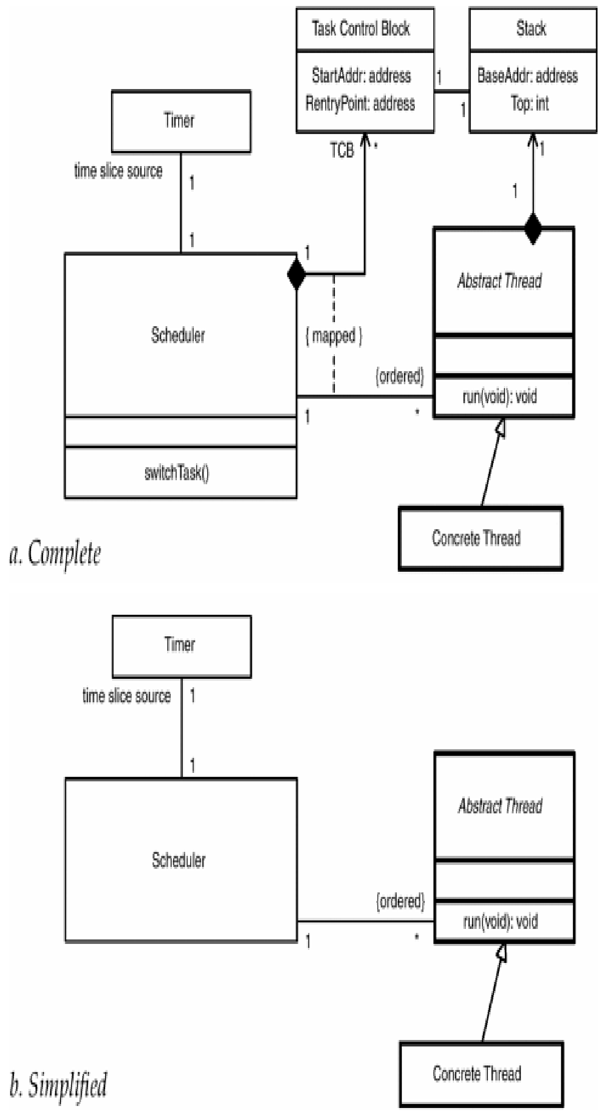

Figure 2. Round Robin Pattern

4.3 Static Priority Pattern

In a static system, a task's priority is determined at compile time and is not changed during execution. This has the advantages of simplicity of implementation and simplicity of analysis.

Figure 3. [4] shows the basic structure of the pattern. Each «active» object (called *Concrete Thread* in the figure) registers with the *Scheduler* object in the operating system by calling *createThread* operation and passing to it, the address of a method defined. Each *Concrete Thread* executes until it completes (which it signals to the OS by calling *Scheduler::return()*), it is preempted by a higher-priority task being inserted into the *Ready Queue*, or it is blocked in an attempt to access a *Shared Resource* that has a locked *Mutex* semaphore.

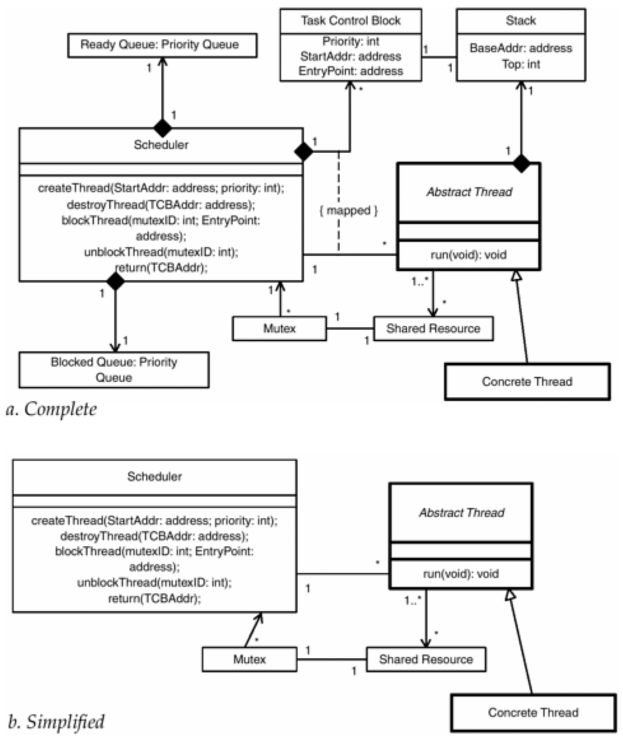

Figure 3. Static Priority Pattern

4.4 Semi-static Priority - Priority Ceiling Pattern

The Priority Ceiling Pattern, or Priority Ceiling Protocol (PCP) as it is sometimes called, addresses both issues of bounding priority inversion (and hence bounding blocking time) and removal of deadlock. It is a relatively sophisticated approach, more complex than the previous methods. It is not as widely supported by commercial RTOSs, however, and so its implementation often requires writing extensions to the RTOS. Figure 4. [4] shows the Priority Ceiling Pattern structure.

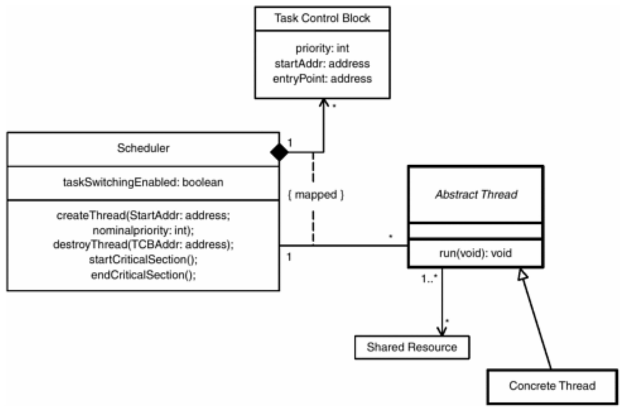

Figure 4. Priority Ceiling Pattern

4.5 Dynamic Priority Pattern

The Dynamic Priority Pattern is similar to the Static Priority Pattern except that the former automatically updates the priority of tasks as they run to reflect changing conditions. There are a large number of possible strategies to change the task priority dynamically. The most common is called *Earliest Deadline First*, in which the highest-priority task is the one with the nearest deadline. The Dynamic Priority Pattern explicitly emphasizes *urgency* over *criticality*.

Figure 5.[4] shows the Dynamic Priority pattern structure.

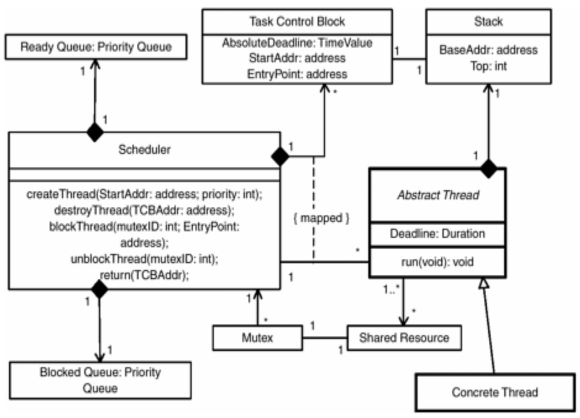
Task Control Block
AbsoluteDeadline: TimeValue
StartAddr: address
EntryPoint: address
Stack
BaseAddr: address
Top: int
Ready Queue: Priority Queue
Scheduler
createThread(StartAddr: address; priority: int);
destroyThread(TCBAddr: address);
blockThread(mutexID: int; EntryPoint: address);
unblockThread(mutexID: int);
return(TCBAddr);
{ mapped }
Abstract Thread
Deadline: Duration
run(void): void
Mutex
Shared Resource
Blocked Queue: Priority Queue
Concrete Thread
1
*
1..*

Figure 5. Dynamic Priority Pattern

V. Tool Support

It is proposed to develop a GUI based tool, which will help the users in selecting appropriate scheduling strategy depending upon the nature and parameters of the real-time tasks in the application being developed. The tool will educate the users about merits and demerits of each scheduling strategy and help in identifying appropriate design pattern. Further, the tool will also generate the code for the chosen design pattern in real-time java. Thus, the design and development activities may be completed in less time and with more accuracy, as the code is generated automatically.

Conclusions

In this paper, we have reviewed and summarized various scheduling approaches for real-time systems. Further, we have listed suitable design patterns for these varied scheduling strategies. The proposed tool to generate code for a chosen design pattern will greatly help the developer in reducing development effort, reducing development time and provides ready-made code, which can be plugged into the developer's application.

References

[1] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
[2] Jane W.S. Liu, *Real-Time Systems*, Pearson Education 2006.
[3] Bruce Powel Douglass, White Paper on *Real-Time Design Patterns* 1998
[4] Bruce Powel Douglass *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*, Addison Wesley 2002
[5] Stankovic, J., M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems*, Norwell, MA: Kluwer Academic Press, 1998.
[6] Douglass, Bruce Powel. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Reading, MA: Addison-Wesley, 1999.
[7] *A System of Patterns: Pattern-Oriented Software Architecture* Buschmann, et. al., John Wiley & Sons, 1996.