Real-time Priority Scheduler

Author: Egon Wuchner, egon.wuchner@siemens.com
Corporate Technology, Siemens AG;
Munich, Germany

The *Real-time Priority Scheduler* architectural pattern concerns real-time systems and introduces several scheduling classes representing different criticality levels for all activities of modern real-time applications. Scheduling is done according to the activities’ priorities and scheduling policies having to guarantee the execution of the most critical activities first and some fairness criteria, too. Furthermore, the Scheduler is responsible for dispatching a newly started, even more critical activity by pre-empting the running, less eligible activity.

Also Known As: Priority Dispatcher

Example: An Assembly Line is controlled by an Assembly Line Manager being responsible for accepting order requests by other parts of the IT business infrastructure. In addition it has to cause the assembly line to process the order within a limited time in order to guarantee a certain throughput of orders. An order specifies the amount of products to be constructed and the three parts that make up the product, a basic, bottom part and two secondary pieces to be placed on the initial one.

The Assembly Line consists of two robots each equipped with a sensor. Each robot is responsible for placing a further piece on the passing basic part being put on the assembly separately on a regular basis. Robots react on sensor events indicating the need to place the picked piece on the passing part without stopping the assembly, thus within a certain time frame.

As a start it seems appropriate to make use of the Rate Monotonic Scheduling algorithm (RMS) at least with respect to the robots’ tasks of adding their picked part to the passing product on the conveyor belt. The algorithm deals with periodic activities with specified execution times and hard deadlines which have to be met under any circumstances. It assigns a certain priority to each of these input tasks which have to be processed in a pre-emptive way in the order of their priorities. Besides, both robots send regular status information to a remote Monitoring Service displaying this information on the screen.

Context: Real-time systems hosting complex real-time applications consisting of different tasks ranging from core, hard real-time activities to normal, less critical responsibilities like coordination or communication with other execution nodes. The Real-time Priority Scheduler pattern describes parts of a real-time system infrastructure below the application level.
Problem

Normally more complex real-time applications consist of several activities being quite heterogeneous with respect to the significance of meeting time-constraints. There is a core of hard real-time activities with certain deadlines after being started whose success is an essential part of the application’s correctness. In contrast functional tasks without any time implications make up the other extreme side within the criticality range an application might have to cover. Last but not least there is the need of activities in order to catch software and hardware interrupts of which an important one for a real-time system might be a regular and fine-grained clock interrupt.

Activities of different criticality levels have to be treated in a separate way. For example a hard real-time activity is compatible to any time-slicing techniques only to a limited extent because it has to run until completion in order to meet its time constraint. The more demanding the time-constraints of activities are, the more restrictive the scheduling strategy has to be.

In addition, there is only one CPU (at least a limited number) on a system, thus only one ready activity of the whole variety of activities can run. The scheduling order of all started activities of a real-time system and its applications always has to reflect their respective time-criticality.

Thus, how is it possible to schedule the whole bunch of activities with different requirements in a criticality-compliant, fair and clear way? The solution of this problem has to resolve the following forces:

- The real-time system has to provide some means of specifying and setting the criticality level (and its options) for each activity. In addition the system should be robust enough to cope with consistent changes of these activity properties.

- Time-critical activities have to take precedence over all deadline independent actions with respect to scheduling. On the other hand the system has to provide some criticality-compatible means of guaranteeing some level of fairness concerning the execution time of activities.

- Starting new activities should imply the proper re-arrangement of the scheduling order according to their criticality level compared to already existing activities either running or being ready to run. In case that the re-arrangement affects the currently running, less critical activity, this activity has to be preempted and the more eligible one has to be dispatched.

- The real-time system itself might contain and start some system mode actions and even normal tasks might enter system mode as soon as calling any system specific functionality like waiting on
I/O. The scheduling model has to consider this type of system activities and has to be open enough in order to integrate new criticality levels without major changes of the whole scheduling mechanism.

- Scheduling has to consider waiting activities which might be blocking on I/O operations or expecting a certain event in order to continue their execution.

- Last but not least the real-time system has to handle interrupts efficiently and in a complementary manner to the scheduling mechanism.

**Solution**

The real-time system introduces several *Scheduling Class* instances representing different categories of activities for each criticality level accepting only certain *Scheduling Policies*. Each Scheduling Class has a set of priorities. Processing activities is done in a pre-emptive way taking Scheduling Classes and their different priorities into account. On the other hand activities of the same Scheduling Class and possibly even of the same priority are scheduled by considering some additional Scheduling Policy.

The real-time system exposes its task entities by an API in order to create and specify the system equivalent of an application’s activity. Each application activity is represented by a schedulable *Task* entity whose state includes the status of the Task like “running”, “ready to run” or “waiting” and its priority.

In detail, the system API offers the flexibility of creating Tasks being members of different *Scheduling Class* strategies according to their different time requirements. It also demands to specify the priority of the Task within its Scheduling Class. Scheduling Class instances make up a hierarchical order with respect to scheduling, i.e. the *interrupt instance* of a Scheduling Class might be the top of the hierarchy and could be followed by a *real-time, a system* and a *user instance*, respectively.

A started Task belonging to a certain Scheduling Class preempts any running Tasks of lower-level Scheduling Class instances of the hierarchy and even those belonging to the same Scheduling Class, but being of lower priority. Besides considering priorities the scheduling within a Scheduling Class is also done according to a certain *Scheduling Policy* dealing with CPU execution time fairness. For instance the Policy of a Real-time Class might allow to associate a specific execution time quantum with certain priority levels each or it might run a Task until completion without being interrupted by Tasks of the same priority.
The higher the level of a Scheduling Class the more restrictive a Scheduling Policy is. The Scheduling Policy of the Real-time Class does not change the priority of its Tasks after pre-emption in order to keep to the real-time priority specified by the application. On the other hand the User Class might boost the priority of a user Task in order to prevent it from starvation. Scheduling Policy specific data of a Task is captured in a special sub-type of a Scheduling Parameter, let it be a Real-time or User Parameter.

Dispatching the most eligible Task lies within the responsibility of a Kernel Dispatcher. Basically the Central Processing Unit executes the instructions of this system dispatcher which delegates program processing by dispatching the most eligible Task. The Kernel Dispatcher also provokes a pre-emption check after each interrupt.

A Kernel Dispatcher finds the Tasks being “ready to run” by its relationship to every Dispatch Queue for each priority of every Scheduling Class instance. Furthermore there are Waiting Queues comprising all Tasks being in a “waiting” state. Each transition of a Task from “running” to “waiting”, e.g. by calling a blocking I/O system operation, results in a wait call to an Event Handle which identifies an event source of the system. Transferring Tasks from a Waiting Queue to the appropriate Dispatch Queue is done by signaling an Event Handle. This implies the transition of the most eligible Task of the Waiting Queue the Event Handle is associated to from “waiting” to “ready to run”. Broadcasting an Event Handle results in signaling all Tasks of its Waiting Queue.

A Task is also allowed to do a waiting call on an Event Handle explicitly in order to be signaled by another Task or even an Interrupt Routine. Each interrupt stops the execution of the currently running Task and causes the Dispatcher to execute the corresponding Interrupt Routine code. Thus processing an interrupt could be parted into a short Interrupt Routine and an interrupt Task. The Interrupt Routine decides which interrupt Task has to process the interrupt and triggers it by signaling the specific Event Handle.

Structure

The Real-time Priority Scheduler architectural pattern has Task entities at its heart. A Task is created and/or run by the real-time system or as an application activity by another activity. The status member of a Task changes from “ready to run” to “running” as soon as it is dispatched and vice versa if pre-empted. The execution context of the Task (i.e. its stack, the code address space, the address of the currently executed program instruction, the CPU register values) has to be saved in case of pre-emption and it has to be restored after resuming the newly dispatched Task. An interrupt implies the suspension of the running Task by saving only few parts of its execution context (e.g. the CPU register values).
The **Scheduling Parameter** base structure to which each Task has a reference contains all scheduling relevant information like the priority. The priority range is determined by the **Scheduling Class** a Task is associated to, either being a real-time, a system or a user Scheduling Class instance. Thus the priority is only valid within the scope of its Scheduling Class.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
<th>Structure</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>• Scheduling Class</td>
<td>• Scheduling Parameter</td>
<td>• Scheduling Policy</td>
</tr>
<tr>
<td><strong>Responsibility</strong></td>
<td>• Scheduling Class</td>
<td><strong>Responsibility</strong></td>
<td>• Scheduling Policy</td>
</tr>
<tr>
<td>• is the schedulable entity of the real-time system</td>
<td>• contains the priority of the Task</td>
<td>• supports queuing new Tasks</td>
<td>• Scheduling Policy</td>
</tr>
<tr>
<td>• resumes its stored execution context when dispatched</td>
<td>• base structure of all parameter sub-types</td>
<td>• delegates the call of the clock-tick interrupt Task to the specific Scheduling Policy</td>
<td>• Scheduling Parameter</td>
</tr>
<tr>
<td>• saves its execution context on pre-emption</td>
<td></td>
<td>• delegates pre-emption of a Task to the Scheduling Policy</td>
<td></td>
</tr>
</tbody>
</table>

The Scheduling Class instances build up a hierarchy. Consequently the running Task belongs to the highest-level Scheduling Class instance. Besides its priority is the highest among all existing Tasks of its Scheduling Class. Each Scheduling Class object is related to all **Dispatch Queues** of its priority levels. Starting a new Task implies queuing it by its Scheduling Class into the appropriate Dispatch Queue. A Dispatch Queue comprises all Tasks being “ready to run” of the same priority of a Scheduling Class.

The Task Class is responsible for offering an interface that supports the scheduling of its related Tasks. First it provides an interface method to queue a new Task. Secondly, it is called by the interrupt clock tick Task in order to process time related properties of the running Task. Depending on the Scheduling Class strategy the time quantum of the Task might have expired which should provoke a pre-emption. The pre-emption of a running Task is also supported by the Scheduling Class in its interface.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
<th>Structure</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduling Class</td>
<td>• Dispatch Queues</td>
<td>• Scheduling Policy</td>
<td>• Scheduling Parameter</td>
</tr>
<tr>
<td><strong>Responsibility</strong></td>
<td>• Scheduling Policy</td>
<td><strong>Responsibility</strong></td>
<td></td>
</tr>
<tr>
<td>• supports queuing new Tasks</td>
<td>• creates special Scheduling Parameter object when queuing a new Task</td>
<td>• delegates the call of the clock-tick interrupt Task to the specific Scheduling Policy</td>
<td>• Scheduling Policy</td>
</tr>
<tr>
<td>• delegates the call of the clock-tick interrupt Task to the specific Scheduling Policy</td>
<td>• facilitates some time execution fairness</td>
<td>• delegates pre-emption of a Task to the Scheduling Policy</td>
<td>• Scheduling Parameter</td>
</tr>
<tr>
<td>• delegates pre-emption of a Task to the Scheduling Policy</td>
<td>• resolves Dispatch Queue position in case of a new Task or in case of queuing a pre-empted one</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Handling these clock-tick and pre-emption calls is done by delegation of the Scheduling Class instance to its specific **Scheduling Policy**. The Scheduling
Policy represents a common interface indicating the Policy’s responsibility for the sequence order of Tasks within some or all Dispatch Queues related to this Scheduling Class. This responsibility concerns queuing new or pre-empted Tasks and creating the special Scheduling Parameter sub-type of a new Task. Another core aspect of a Scheduling Policy is to guarantee some level of execution fairness by adapting the properties of the concrete Scheduling Parameter instance of a running Task and initiating the pre-emption of the Task if necessary.

For instance the Real-time Parameter of a real-time Task contains a time-quantum field allowing to specify a maximum execution time for a real-time Task and a time-left field to measure the executed time. Enforcing or ignoring the time-quantum is done by the Scheduling Policy. The Round-Robin Policy checks the execution time quantum after a clock-tick and decrements the time-left field. In case of time-left value of zero it initiates the pre-emption of the Task in order to continue the execution of the next Task having the same priority and belonging to the same Scheduling Class - unless no more eligible Task has entered the system. In comparison the First-In-First-Out Policy runs a real-time Task without pre-emption unless there is no more eligible Task available.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
<th>Class</th>
<th>Collaborators</th>
</tr>
</thead>
</table>
| Responsibility      | • enforces execution time fairness for all Tasks of the same priority  
|                     | • maintains the priority of a real-time Task                          | Responsibility         | • protects a running Task from being pre-empted by Tasks of the same priority |

As for User Policies used by the user Scheduling Class instance each User Policy assigns a time-quantum to a User Parameter of a user Task. Furthermore a special Boost Policy might change the priority of the Task. It could decrease the priority after time-quantum expiration in order to enhance the execution chance of other user Tasks. On the other hand it may boost the priority of the running user Task after needing more than a maximum time value to achieve its time-quantum in order to protect the Task from starvation.

As a last example let us consider a system call of a user Task. Such a call might be accompanied by a trap interrupt. This interrupt could call the User Class of the Task and its Policy could switch the Task to a higher-level system Scheduling Class instance during this system call. As a consequence
it could change the user priority to a system priority which might be part of a User Parameter. This actions aim at completing a system call as soon as possible.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Dispatcher</td>
<td></td>
</tr>
<tr>
<td><strong>Responsibility</strong></td>
<td>• Task</td>
</tr>
<tr>
<td></td>
<td>• Dispatch Queue</td>
</tr>
<tr>
<td></td>
<td>• creates hierarchy of Scheduling Class instances</td>
</tr>
<tr>
<td></td>
<td>• dispatches most eligible Task</td>
</tr>
<tr>
<td></td>
<td>• pre-empts the running Task if necessary</td>
</tr>
<tr>
<td></td>
<td>• invokes the interrupt routine in case of an interrupt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatch Queue</td>
<td></td>
</tr>
<tr>
<td><strong>Responsibility</strong></td>
<td>• Task</td>
</tr>
<tr>
<td></td>
<td>• Kernel Dispatcher</td>
</tr>
<tr>
<td></td>
<td>• queues Tasks being “ready to run”</td>
</tr>
<tr>
<td></td>
<td>• contains all Tasks of a certain priority of a Scheduling Class instance</td>
</tr>
</tbody>
</table>

The scheduling effect of a Scheduling Policy is only considered within the scope of a Scheduling Class. Considering the hierarchy of Scheduling Class objects by dispatching the most eligible Task is done by the Kernel Dispatcher. The Dispatcher is responsible for creating the Dispatch Queues and the Scheduling Class instances. Dispatch Queues comprise all Tasks being “ready to run” whereas a certain Dispatch Queue is always referred to by a single Scheduling Class instance containing all Tasks of the same priority.

Hence the Dispatcher knows the Dispatch Queues of all Scheduling Classes and accesses them according to the hierarchy of Scheduling Class objects and the priorities within a Scheduling Class. The Dispatcher tests for pre-emption after an interrupt, suspends and pre-empts the interrupted Task if needed and searches its hierarchical sequence of Dispatch Queues in order to dispatch the most eligible Task next.

In contrast, Waiting Queues encompass all “waiting” Tasks and are used by Event Handles to queue all Tasks waiting for the corresponding event.
A Waiting Queue takes the Scheduling Class instances and priorities of its Tasks into account in order to reflect the correct scheduling order of Tasks.

Each Event Handle has a relationship to a Waiting Queue which contains all Tasks waiting or blocking for the event the Event Handle stands for, i.e. an input/output event (I/O) or an interrupt. The Event Handle provides the means to a Task to wait or block for this event and to be signaled in case of its occurrence. Signaling the next eligible Task or broadcasting all waiting Tasks of the Waiting Queue of an Event Handle can be done by any running Task which knows the given Event Handle by some means. This could also be an Interrupt Routine being a piece of code whose execution is started by the Dispatcher in case of certain interrupts.

The following class diagram illustrates the participants of the Real-time Priority Scheduler pattern and their relationships.
Dynamics

One of the most important actions is to pre-empt a running Task in order to resume a more eligible Task being ready to run. There are two possibilities of originating a pre-emption: testing for the need to pre-empt after having run an Interrupt Routine because of an interrupt or after having created a more eligible Task that should replace the execution of the current one.

First, the following three scenarios deal with the actions after a clock-tick interrupt has occurred. It comprises the interrupt processing and the pre-emption of the running Task by the Kernel Dispatcher.

**Scenario 1** shows how the clock-tick interrupt processing is separated into a corresponding Interrupt Routine and an interrupt Task which is put from a “waiting” state into a “ready to run” state. Being more eligible than the interrupted Task it will be dispatched in Scenario 2.
Scenario 1 consists of the following collaboration steps:

- The incoming clock-tick interrupt makes the Kernel Dispatcher stop running the current Task and to suspend its execution state. The next step is to run the relevant Interrupt Routine.

- The Interrupt Routine just signals the appropriate Event Handler object. The Event Handler dequeues the most eligible Task of its associated Waiting Queue (i.e. the interrupt Task) and queues the Task into the Dispatch Queue related to the Scheduling Class and the priority of the Task.

Scenario 2 continues the first one after having called the clock-tick Interrupt Routine. Since the more eligible interrupt Task is “ready to run” the Kernel Dispatcher test method “to_preempt” results in pre-empting the previously running Task and dispatching the clock-Task.
There are the following steps of the message sequence flow:

- Returning from the Interrupt Routine the Kernel Dispatcher tests if a more eligible Task than the interrupted Task has entered the system. This is done by searching the sequence of Dispatch Queues according to the hierarchy involved by all Scheduling Class objects and priorities. As the Interrupt Routine has triggered a corresponding, more eligible interrupt Task the former Task has to be pre-empted.

- After retrieving the Scheduling Class of the interrupted Task the Kernel Dispatcher delegates the pre-emption to the Scheduling Class instance.

- The Scheduling Class resolves the right Dispatch Queue position for the interrupted Task and eventually queues it.
• In the end the Kernel Dispatcher dispatches the most eligible Task being ready to run and resumes its execution next. This could be the interrupt Task in case of currently being the most eligible Task “ready to run” of all Dispatch Queues.

Thus **Scenario 3** shows the clock interrupt Task decrementing the time left for execution of the interrupted Task which is a real-time Task of a Round Robin Policy.

• The Kernel Dispatcher requests the Scheduling Class instance of the running Task and delegates the clock-tick call to the returned real-time Scheduling Class object.
• The real-time Scheduling Class object retrieves the Scheduling Parameter of the Task and passes it to its Scheduling Policy in order to react according to the fairness criterion imposed by the Round Robin Policy.

• The Round-Robin Policy decrements the left execution time of the interrupted Task and tests for the need of pre-emption by fetching the previously changed execution time and comparing it to zero.

• As the time left for execution is zero the interrupted Task is moved to the end of the containing Dispatch Queue. The interrupt Task returns and the Kernel Dispatcher dispatches the next eligible Task.

All previous scenarios deal with interrupt processing being connected to the pre-emption of an interrupted Task. In contrast Scenario 4 shows how a Task creates a more eligible Task and how the creating Task yields execution to the new one.

• A running Task creates another Task by specifying the Scheduling Class level, the priority and the specific Scheduling Policy.
• Creating a Task results in resolving the right Dispatch Queue and the Dispatch Queue position in order to queue the new Task.

• The last steps in starting a new Task compares the Scheduling Class objects and if necessary the priorities of the running and the created Task by calling the “to_preempt” method of the Kernel Dispatcher. If the created Task is more eligible than the running one the Kernel Dispatcher switches the execution by saving the context of the old Task and dispatching the next more eligible Task.

The last scenario partially depicts the collaboration in case of changing any Task properties as well. Examples could be to change the priority of a Task within the same Scheduling Class, switch to another Scheduling Class and priority and even to apply another Scheduling Policy within the given Scheduling Class of a Task.

Implementation

Mainly, there are two application areas of this pattern. The first one is to implement a real-time operating system according to the concept of this pattern. The other is to build a real-time system atop a basic, simplistic real-time operating system which only offers a range of priorities and schedules its tasks in a pre-emptive, priority-sensitive way.

This implementation section sketches the steps for the second implementation option using C++. Thus we assume that the underlying real-time operating system provides some OS::Task, OS::EventHandle, OS::WaitingQueue, OS::InterruptRoutine (wrapper) classes and a range of 99 priorities in an decreasing eligibility order. Internally it has some sort of Kernel Dispatcher on which we can rely with respect to dispatching, pre-emption and interrupt processing. Nevertheless we need our own Dispatcher and Dispatch Queues in order to enforce Scheduling Class strategies and Scheduling Policies.

1 Define your own Task class and combine it with the given OS::Task: We use an OS::Task instance as a private member of our Task class. Using delegation instead of derivation aims at preventing the application from having direct access to the OS. As soon as the OS::Task member is started it is scheduled by the underlying pre-emptive OS.

The Task interface should contain some means to be created, run and suspended. Suspending the Task is done by making the currently running OS::Task member yield the execution to the next eligible OS::Task. That OS::Task is related to another Task.

Saving and resuming its execution (context) is automatically done by the underlying operating system because of linking each Task to an OS::Task.
class Task
{
public:
    Task(Sched_Class::Class_Enum sched_class,
         Priority sched_priority,
         Sched_Class::Policy_Enum sched_policy );
    virtual ~Task();

    // running and suspending the Task
    virtual void run() = 0;
    void suspend() { p_os_task_->yield(); };

    // changing scheduling properties
    void change_sched_class(
            Sched_Class::Class_Enum sched_class,
            Priority sched_priority);
    void change_sched_policy(Policy policy);

    // accessor methods
    Sched_Param* const get_sched_param() const;
    Sched_Class* const get_sched_policy() const;
    Priority get_priority() const;

private:
    void run_impl() { this->run();
        p_sched_class_->dequeue(this) };

    OS::Task              os_task_; 
    Sched_Param*  p_sched_param_; 
    Sched_Class*  p_sched_class_; 
};

The application has to derive from this Task class and implement the “run” method. This method provides the code this Task is intended to execute. The interface also allows to change the scheduling properties. The Task has a one-to-one relationship to its Scheduling Parameter object whose special sub-type is consistent to the specific Scheduling Class. The constructor implementation shows this in more detail.

Task::Task(
            Sched_Class::Class_Enum sched_class,
            Priority sched_priority,
            Sched_Class::Policy sched_policy )
: os_task_( this->run_impl )
{
    // set Scheduling Class
    p_sched_class_ = Sched_Class::get_class( sched_class,
                                             sched_policy);
    // create the SchedulingParameter object
    p_sched_param_ = sched_class->create_sched_param(priority);
    // queue the Task to the appropriate DispatchQueue
    p_sched_class_->queue(this);
    // let Dispatcher find out about possible pre-emption
    Dispatcher::instance()->to_preempt(this);
    // start OS::Task, OS will perform pre-emption of
    // currently running Task and dispatch this one
    // if needed
    os_task_.start();
}
At last the OS::Task is started after having passed the method pointer of run_impl at initialization time. The run_impl method calls run. After run returns which means the end of the OS::Task it has to clean up the Task by removing it from the Dispatch Queue of its Scheduling Class.

2 Define the Dispatch Queue interface and implement the access efficiently: The Dispatch Queue interface provides some iterator functionality. In addition it has to facilitate the insertion of Tasks at certain positions of the queue.

class Dispatch_Queue
{
public:
   // enumerating queue position options
   enum Position {Begin=0, End=1};

   // initial length as parameter
   DispatchQueue( unsigned long number_of_Tasks);
   ~DispatchQueue();

   // iterator functionality
   Task* first();
   Task* next();
   void queue_at_position( Task* p_sched, Position pos );

   // dequeue first or certain Task
   Task* dequeue();
   void dequeue( Task* p_sched );
};

In implementing this interface we could use double-linked lists, hash tables or a combination of both to diminish the time needed to access the queue.

3 Specify the hierarchy of Scheduling Class instances. This task consists of specifying the number of Scheduling Class instances within the real-time system, identifying them by name and determining a hierarchy order. The Kernel Dispatcher could provide a static interface in order to create/get a specific Scheduling Class instance of the hierarchy.

Additionally specify the type and the range of priorities for each Scheduling Class instance. A simple approach might be to have a global type of priorities and to partition the OS range of priorities into a set of ranges, one for each Scheduling Class. For example we could determine a hierarchy of interrupt, real-time and user Scheduling Class instances each level using a subsequent range of 33 operating system priorities.

4 Define the Scheduling Class and determine its instances: Besides each Scheduling Class level (interrupt, real-time and user) we have to figure out the number of Scheduling Class instances per level. For example each Scheduling Policy a Scheduling Class level supports might imply a separate
Scheduling Class instance. Thus we come up with two real-time Scheduling Class instances for the First-In-First-Out and the Round-Robin Policy.

class Sched_Class
{
public:
    // an interrupt instance
    Sched_Class( Sched_Policy::Interrupt_Policy );

    // a real-time instance
    Sched_Class( Sched_Policy::Real_Time_Policy );

    // a user instance
    Sched_Class( Sched_Policy::User_Policy );

    ~Sched_Class();

    Dispatcher::Class_Enum get_hierarchy_level() const
    { return level_; };

    // Dispatcher has to set the Dispatch Queue array
    void set_dispatch_queue(
        Dispatch_Queue[]* p_dispatch_queues )

The interface of the Scheduling Class has to support the pre-emption process, handling the clock-tick interrupt and queuing a newly created Task. This is done mainly by delegating the messages to the Scheduling Policy it is related to.

    void queue( const Task* p_sched )
    {
        Priority priority = p_sched->get_priority();
        Dispatch_Queue* p_queue = (*p_dispatch_queues)[priority];

        Dispatch_Queue::Position pos =
            p_sched_policy_ -> resolve_queue_position( p_sched, p_queue );
        p_queue->queue_at_position( pos, p_sched );
    }

    void preempt( Task* p_sched ) {
        // similar to queue
    }

    void clock_tick( Task* p_sched )
    {
        Sched_Param* p_sched_param = p_sched->get_sched_param();
        p_sched_policy_ -> clock_tick( p_sched_param );
    }

private:
    // identifying the scheduling class instance
    Dispatcher::Class_Enum identity_;

    // policy this scheduling class instance is related to
    Sched_Policy* p_sched_policy ;

    // array of dispatch queues, set by Dispatcher
    Dispatch_Queue** p_dispatch_queues_;
5 Construct the Scheduling Policies: This task implies following steps:

5.1 Find a way to identify Scheduling Policy instances: When creating Scheduling Class instances we need access to the Scheduling Policies they support. The simplest solution is to have exactly one Policy instance for each specific Policy class. Consequently, we could specify several static methods of the Sched_Policy base class which allow to retrieve these instances:

```cpp
class Sched_Policy
{
public:
    enum Interrupt_Policy { Default=0 };  
    enum Real_Time_Policy { FiFo=0, RoundRobin=1 }; 
    enum User_Policy { Default=0, Boost=1 }; 

    static Sched_Policy* get_interrupt_policy();
    static Sched_Policy* get_realtime_policy(
        Real_Time_Policy rt_policy );
    static Sched_Policy* get_user_policy(
        User_Policy user_policy );
};
```

An alternative might be to move these static methods to the specific Sched_Policy subtypes. This prevents the Sched_Policy base class from having to know all its subtypes.

5.2 Specify the interface of the base Scheduling Policy class:

The Scheduling Policy interface has to provide an interface which deals with actions which might have implications on the currently running Task. This includes handling the clock-tick interrupt, resolving the queue position in case of pre-empting the current Task and creating the special Scheduling Parameter of a new Task.

```cpp
void resolve_queue_position( Task* p_sched,
                Dispatch_Queue* p_queue ) = 0;

clock_tick( Sched_Param* p_sched_param ) = 0;

Sched_Param* create_sched_param( Priority priority ) = 0;
```

5.3 Define the Scheduling Policy subtypes:

Define the Scheduling Policy sub-types for each Scheduling Class instance. These are the First-In First-Out Policy and Round-Robin Scheduling Policy for the real-time Scheduling Class instance. On the other hand we have to define a Boost Policy for the user Scheduling Class instance. The implementations of the abstract methods of Sched_Policy might look like the following for a Round-Robin Policy:
class Round_Robin_Policy
{
public:
clock_tick( Sched_Param* p_sched_param,
        Dispatch_Queue* p_queue_of_sched )
{
    Real_Time_Param* p_rt_param =
        dynamic_cast<Real_Time_Param*>(p_sched_param);
    p_rt_param->decrement_time_left();
    // do we need to pre-empt currently running Task?
    bool is_time_zero = p_rt_param->check_preemption();
    if( true == is_time_zero ) {
        Task* p_sched = p_current_queue->dequeue();
        p_queue_of_sched->queue_at_position(
            p_queue_of_sched->End);
    }
}
};

Sched_Param* create_sched_param( Priority ) {
    return new Real_Time_Param( Priority );
};

Dispatcher::Position
resolve_queue_position( Task* p_sched,
        Dispatch_Queue* p_queue )
{
    // check if Task is part of queue,
    // if no return Dispatcher::End
};

6 Specify the responsibilities of the Dispatcher and implement them. The Dispatcher is a typical singleton within the whole system. Implementing the responsibilities of a Dispatcher involves several steps:

6.1 Creating and managing the Dispatch Queues: The Dispatcher is responsible for creating and storing all Dispatch Queues for each priority within the range of 0..99. An appropriate solution seems to be to separate the Dispatch Queue array according to Scheduling Class levels.

class Dispatcher
{
public:
    Dispatcher()
    {
        p_interrupt_queues_  = new Dispatch_Queue[33];
        p_rt_queues_         = new Dispatch_Queue[33];
        p_user_queues_       = new Dispatch_Queue[33];
        ...
    }
};

6.2 Creating and managing the Scheduling Class instance hierarchy: There are three Scheduling Class levels and each implies its own set of Scheduling Class instances. Each level has a separate Scheduling Class instance for each Scheduling Policy the Scheduling Class might support.
One implementation might be to distinguish Scheduling Class levels and instances within a level by overloaded static get methods with a parameter specifying the Scheduling Policy.

```cpp
enum Class_Enum { interrupt = 0, real_time=1, user=2 };
static Sched_Class* get_sched_class(Sched_Policy::Interrupt_Policy);
static Sched_Class* get_sched_class(Sched_Policy::Real_Time_Policy);
static Sched_Class* get_sched_class(Sched_Policy::User_Policy);
```

6.3 Pre-empting currently running Task: Our high-level Dispatcher is a kind of mediator between the OS dispatcher and the Scheduling Class, its Dispatch Queues and Scheduling Policies we introduce atop the OS. Thus the Dispatcher has a member pointing to the currently running Task.

In case of pre-emption its “to_preempt” method makes the current Task be queued at the relevant position of the appropriate Dispatch Queue. Furthermore it has to cause the OS::Task beneath this Task to be pre-empted by some means of the underlying OS.

If the OS does not provide such mechanisms we could apply the following simple technique. We halve the original range of 33 priorities within a Scheduling Class instance to a range of 16 priorities. A priority of a Scheduling Class is mapped to two subsequent OS priorities. The OS::Task priority of a running Task is always the lower, more eligible one of the two. Having to pre-empt this OS::Task because of some time criterion (e.g. imposed by a Round-Robin Policy) is done by incrementing the OS priority of this Task, decrementing the OS priority of the next eligible Task of the same Dispatch Queue and setting the Dispatcher member to this new Task.

The “to_preempt” method is called after having created a new Task or by the clock-tick interrupt Task (see constructor of Task of step 1 and interrupt Task of step 9).

6.4 Registering Interrupt Routines on a high-level: The Dispatcher should implement a way of registering one’s own Interrupt Routine method for certain interrupts with the OS. This is necessary in order to signal a certain OS::EventHandle object within a clock-tick Interrupt Routine. The interrupt clock-tick Task waits for that OS::EventHandle and processes the clock-tick with respect to the current Task of the Dispatcher by calling Dispatcher::to_preempt.

7 Specify the Scheduling Parameter base structure: Add access operations to get and set the priority.
8 Define the Scheduling Parameter sub-types: Each Scheduling Class instance is accompanied by a special Scheduling Parameter type.

Especially the state and behaviour of each sub-type is strongly related to the special Scheduling Policies belonging to this Scheduling Class instance. At least define a Real-time Parameter and a User Parameter since the Scheduling Policies of the real-time and the user Scheduling Class differ significantly. The definition of a Real-time Parameter might partially look like this:

```cpp
class Real_Time_Param : public Sched_Param
{
public:
    Time_Value get_time_quantum() const;
    Time_Value get_time_left() const;
    void decrement_time_left();
private:
    ...
};
```

9 Interrupt Routine and interrupt Task to deal with the clock-tick OS interrupt: We define an own Interrupt Routine and register it with the Dispatcher for the clock-tick interrupt. This Interrupt Routine shares an OS::EventHandle with an interrupt clock-tick Task. The clock-tick Task is responsible of enforcing any time related Policy of the running Task by calling the Task’s Scheduling Class instance.

In order to catch the interrupts of the two sensors we introduce two Interrupt Routines using two separate OS::EventHandles. Additionally we create two corresponding interrupt Tasks being of high priority and waiting for the OS::EventHandle to be signaled by the Interrupt Routine.

In turn each interrupt Task triggers another OS::EventHandle which is shared with a real-time Task counterpart. These two real-time Tasks are of the same priority and run the robots in order to place a piece on the passing part of the conveyor belt. Both real-time Tasks are scheduled according a Round-Robin Policy (because of the single CPU system).

Eventually we have two user Tasks, one for each robot, communicating with the Monitoring Service in order to transmit status information. The Monitoring Service itself is a user Task, all of them using a Boost Policy.
The **Real-time Multiple CPUs Priority Scheduler** aims at systems consisting of multiple Central Processing Units (CPUs). Every CPU is represented by a Kernel Dispatcher object, each executing a separate Task in parallel. Thus there are several Kernel Dispatchers within the system with their own set of Dispatch Queues.

Several CPUs add some complexity to pre-empting a Task running on a certain CPU. The pre-empted Task is not constrained to being dispatched again on the same CPU. There is the option of queuing it on the appropriate Dispatch Queue associated to the Kernel Dispatcher of another CPU.

Thus different strategies might apply in choosing another CPU. A pre-empted real-time Task could be queued on the Dispatch Queue of the CPU running the least eligible Task in order to ensure that the real-time Task is placed into execution as fast as possible. On the other hand a user Task might be placed on the Dispatch Queue of the CPU last used in order to reduce bus traffic resulting from cache migration between CPUs.

Hence, these different CPU selection strategies are accompanied by different Scheduling Classes like a **Real-time Scheduling Class** or a **User Scheduling Class** deriving from a base interface represented by the Scheduling Class. As a summary the scheduling strategy concerning all CPUs is done by a concrete subclass of the Scheduling Class, whereas the scheduling strategy affecting the order of Dispatch Queues of a certain Kernel Dispatcher is still implemented within the Scheduling Policy of the Scheduling Class.

The **Solaris Operating System** beginning with version 2.6 is a symmetric multiple pre-empting system and uses Kernel Threads as real-time system Tasks ([SolOS1], [SolOS2], [POSIX]). Furthermore it makes usage of a hierarchy of several Scheduling Classes like an Interrupt, a Real-time, a System and a User Scheduling Class. Each Kernel Thread is related to a certain concrete Scheduling Class which in turn is associated to a certain Scheduling Policy. Since Solaris implements the POSIX standard to a great extent there are the options of a First-In First-Out Policy and a Round-Robin Policy for the Real-time Scheduling Class.

Each CPU is related to its Dispatch Queues and there are Event Handles like Mutexes and Condition Variables each being connected to a dedicated Waiting Queue called Turnstile. The former ones allow to synchronize access to a certain block of program instructions used by several Kernel Threads. The latter ones allow a Kernel Thread to trigger other Kernel Threads in full compliance with the original thought of an Event Handle described above.

The **Windows CE 3.0** operating system is a single CPU pre-empting real-time operating system mostly implementing the Real-time Priority Scheduler pattern shown above ([MSDN00], [MSJ99]). Threads are the
Tasks defined by Windows CE 3.0 and there is the notion of Interrupt Service Threads and Interrupt Service Routines sharing the work of processing interrupts. There is no explicit concept of a Scheduling Class, instead there is a range of Thread priorities between 0 and 255, each priority being represented by a different Dispatch Queue. Windows CE 3.0 provides the option of specifying a time quantum for each Thread. Not setting a time quantum implies the execution of the Thread until completion. Therefore Windows CE 3.0 supports the concept of different Scheduling Policies. There are several Event Handle variants like critical sections, mutexes and events.

The **Real-time Java Runtime Environment** specified by the JConsortium [RTSJ] contains the classes/interfaces Schedulable, SchedulingParameter, Scheduler, PriorityScheduler and DispatchQueues. It is a pre-emptive runtime environment offering an API for real-time applications written in Java on top of a real-time operating system.

The specification prescribes a fixed-priority pre-emptive scheduling mechanism and at least 28 unique priority levels supported by the underlying system. A Schedulable is related to its Scheduler and its specific SchedulingParameter object. As a minimum each implementation has to provide a PriorityScheduler and its relevant PriorityParameter sub-type of the SchedulingParameter class. This PriorityScheduler consistently uses the same First-In-First-Out Policy for each priority.

A special implementation could offer alternative Scheduler strategies as well. The specification does not specify the relationships between different Schedulers. Thus different Schedulers could be built along the concept of Scheduling Classes described above. Schedulers might represent a Scheduling Classes dividing the range of 28 priorities into sub-ranges and building up a hierarchy that is to be considered by the scheduling mechanism of the Real-time Java Runtime Environment.

**Consequences**

The Real-time Priority Scheduler pattern offers at least three benefits:

*Variety of Tasks combined with execution fairness:* The pattern covers different categories of Tasks with respect to execution precedence because of different time requirements. Each criticality level is represented by a Scheduling Class instance (or a separate sub-type of Scheduling Class). Within each Scheduling Class a Scheduling Policy sub-type enforces a certain degree of execution fairness. The higher-level the Scheduling Class instance is, the more restrictive the Scheduling Policy might be with respect to time slicing and priority changing.

*Transparent and robust scheduling:* Scheduling is performed in a transparent, clear and robust way. In creating Tasks the application developer only has to specify such properties as the Scheduling Class, the priority and maybe the Scheduling Policy. Afterwards he/she does not need
to care about execution matters except for changing those properties if needed.

Modifying any of these properties of a running Task yields in testing the new eligibility and pre-empting it if needed. Resetting this scheduling information of a Task being “ready-to-run” causes the Task to be correctly queued to another Dispatch Queue.

Scheduling and Dispatching is done according to some clear principles: Tasks are dispatched due to the hierarchy of Scheduling Class instances and the priorities within a Scheduling Class. Creating and starting a new, more eligible Scheduling leads to the pre-emption of the running Task and dispatching the new one.

**Extensibility:** The open structure of the Real-time Priority Scheduler offers the option of adding new Scheduling Class instances/sub-types within a given hierarchy. The pattern is also prepared for introducing and using further Scheduling Policies if needed.

**Interrupt Handling:** Interrupts processing is aligned along the concept of Scheduling Classes and pre-emption. The Interrupt Routine signals the interrupt Task by using an Event Handle which does all significant work. This keeps the real-time system highly sensitive to newly arriving interrupts because of the minimal CPU seize of Interrupt Routines. Furthermore the notion of Event Handles and Waiting Queues supports implementing blocking I/O operations and synchronization objects like mutexes and condition variables.

The Real-time Priority Scheduler pattern has the following **liabilities**:

**Degree of Complexity:** There is a certain inherent complexity reflected by many classes and a high degree of interaction.

Scheduling Classes and Policies take care of scheduling and pre-emption mechanisms. On the other hand Waiting Queues, Event Handles and Interrupt Routines are dealing with interrupt processing. These separate functionalities interact because of time execution fairness and Event Handles. The different responsibilities make up the complexity and extended interaction.

**Priority Inversion:** Even implementing synchronization objects like mutexes as Event Handles by maintaining the waiting Tasks within the associated Waiting Queue does not prevent the possibility of priority inversions. A less eligible Task might have acquired the mutex but has been pre-empted in the meantime. A more eligible Task trying to acquire the mutex as well will fail and has to enter the Waiting Queue. As a consequence it releases the CPU to a less eligible Task.
Limited scheduling algorithms applicability: the real-time scheduling pattern is only applicable for certain scheduling algorithms like Rate-Monotonic Scheduling. This scheduling technique heavily relies on priorities. Other algorithms like Earliest Deadline First (EDF) or Minimum Laxity First (MLF) deal with time criteria and can not be directly mapped to the mechanisms of a Real-time Priority Scheduler. There needs to be a scheduling system above the Real-time Priority Scheduler which translates the scheduling properties like time data and the actual time to Scheduling Classes, priorities and Scheduling Policies.

See Also

The relationship of the Scheduling Class and its associated Scheduling Policy is an application of the Strategy pattern [GoF94]. Additionally the Strategy pattern also applies to the Task and its Scheduling Class in case of multiple Scheduling Class sub-types.

The Scheduling Policy, its sub-classes, Scheduling Parameter and those sub-types relate to the Factory pattern [GoF94]. Each specific Scheduling Policy type creates a certain Scheduling Parameter instance for each new Task.

The Kernel Manager is a manager [Som] of its Dispatch Queues by administering them according the Scheduling Classes and priorities they correspond to.

Credits

Thanks to my shepherd, Dietmar Shuetz, he invested plenty of patience, time and thinking going through all the versions of this paper. Some special thanks to my colleague Christa Schwanninger who encouraged me all the time to keep writing, even in the worst situations of doubt. And last but not least I would like to thank my wife Karin. Without her comprehension for such an interesting and time-consuming issue as pattern writing I wouldn’t have completed this pattern submission to EuroPLoP.

References

[GoF94] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software; Addison-Wesley 1994


Handouts of the course: “Solaris 8 Operating Environment Internals”
Module 2, “Multithread Architecture”;
Sun Microsystems, Inc.

David R. Butenhof:
Programming with POSIX Threads;
Addison Wesley 1997

Bollela, Gosling, Bros gol, Dibble, Furr, Hardin, Turnbull:
The Real-Time Specification for Java;
Sun Microsystems 2000
http://www rtj.org

Paul Yao:
Windows CE 3.0: Enhanced Real-time Features Provide Sophisticated Thread Handling;
Microsoft Software Developer Network Magazine (MSDN), November, 2000

Douglas Boling:
Updated with New Kernel Features, Windows CE 3.0 Really Packs a Punch;
Microsoft Journal (MSJ), July, 1999

D. Schmidt, M. Stal, H. Rohnert, F. Buschmann:
Pattern Oriented Software Architecture, Patterns for Concurrent and Networked Objects;
John Wiley & Sons