S.Ha.R.K. User Manual

Scuola Superiore di Studi e Perfezionamento S. Anna
ReTiS Lab

Volume III

S.Ha.R.K. MODULES

Written by
Paolo Gai (pj@sssup.it)

RETIS Lab.
Scuola Superiore S. Anna
Via Carducci, 40 - 56100 Pisa

16th December 2004

Contents

Abstract

The S.Ha.R.K. kernel provides a way to implement in a simple way a lot of scheduling algorithms that have been proposed by the Real-Time literature. This Volume simply contains a simple description on which modules are available and how these modules can be used. Moreover, to help the implementation of new modules, a small description of the tricks used in the implementation is also given.

Chapter 1

Models

Task Models and Resource Models are the structures used by the S.Ha.R.K. Kernel to isolate the scheduling parameter needed by the various Scheduling Modules and Resource Modules.

To simplify the use of the Models, the Kernel provides a set of macros that can be used to fill their various fields. In the following paragraphs, the various models are described in detail. All the Model definitions of this chapter can be found in the file `include/kernel/model.h`.

Examples of Models initializations can be found in various examples through this Volume.

1.1 HARD_TASK_MODEL

```
typedef struct {
   TASK_MODEL t;
   TIME mit;
   TIME drel;
   TIME wcet;
   int periodicity;
   } HARD_TASK_MODEL;
```

A Hard Task model can be used to model periodic and sporadic tasks. These tasks are usually guaranteed basing on their minimum interarrival time (mit) and wcet, and may have a relative deadline.

Table 1.1 shows the macros that applies to a `HARD_TASK_MODEL`.

1.2 SOFT_TASK_MODEL

```
typedef struct {
   TASK_MODEL t;
   TIME period;
   TIME met;
   TIME wcet;
   int periodicity;
   int arrivals;
   } SOFT_TASK_MODEL;
```

Macro name	Behaviour
hard_task_default_model(m)	Default values for the Model (periodic task, others = 0)
hard_task_def_level(m,l)	Set the Model level to l. A Module registered at level x can accept a Model only if it has level 0 or x. The default value is 0.
hard_task_def_arg(m,a)	Set the void * argument passed to the task the first time it is activated. The default value is NULL.
hard_task_def_stack(m,s)	Set the task stack size. The default value is 4096.
hard_task_def_stackaddr(m,s)	If the stack is statically allocated you can tell its address here. The default is NULL (no pre-allocated stack).
hard_task_def_group(m,g)	Set the task group to g. Task grouping influences primitives like group_activate() or group_kill()
hard_task_def_usemath(m)	Declare that the task uses floating point arithmetic.
hard_task_def_system(m)	Declare that the task is a system task. System tasks behaves differently at shutdown. See the Architecture manual for details.
hard_task_def_nokill(m)	Declare that the task can not be killed. These tasks behaves differently at shutdown. See the Architecture manual for details.
hard_task_def_ctrl_jet(m)	If called, the Kernel must store JET informations for the task.
hard_task_def_mit(m,p)	Set the Minimum Interarrival Time (MIT) of the Model to p.
hard_task_def_drel(m,d)	Set the relative deadline to d.
hard_task_def_wcet(m,w)	Set the Worst Case Execution Time to w.
hard_task_def_periodic(m)	Declare that the task is Periodic.
hard_task_def_aperiodic(m)	Declare that the task is Sporadic (Aperiodic).
hard_task_def_joinable(m)	Declare that the task is joinable with task_join()/pthread_join().
hard_task_def_unjoinable(m)	Declare that the task is not joinable (is detached) with task_join()/pthread_join().
hard_task_def_trace(m)	Declare that the task has to be traced by the Tracer.
hard_task_def_notrace(m)	Declare that the task has not to be traced by the Tracer.

Table 1.1: HARD_TASK_MODEL Macros

A Soft Task model can be used to model periodic and aperiodic tasks usually not guaranteed or guaranteed basing on their period and mean execution time (met). A Soft task can also record pending activations if the arrivals are set to SAVE_ACTIVATIONS. A wcet field is also present for those servers that need it (i.e., TBS).

Table 1.2 shows the macros that applies to a `SOFT_TASK_MODEL`.

Macro name	Behaviour
soft_task_default_model(m)	Default values for the Model (periodic task, save arrivals, others = 0).
soft_task_def_level(m,l)	Set the Model level to l. A Module registered at level x can accept a Model only if it has level 0 or x. The default value is 0.
soft_task_def_arg(m,a)	Set the void * argument passed to the task the first time it is activated. The default value is NULL.
soft_task_def_stack(m,s)	Set the task stack size. The default value is 4096.
soft_task_def_stackaddr(m,s)	If the stack is statically allocated you can tell its address here. The default is NULL (no pre-allocated stack).
soft_task_def_group(m,g)	Set the task group to g. Task grouping influences primitives like group_activate() or group_kill()
soft_task_def_usemath(m)	Declare that the task uses floating point arithmetic.
soft_task_def_system(m)	Declare that the task is a system task. System tasks behaves differently at shutdown. See the Architecture manual for details.
soft_task_def_nokill(m)	Declare that the task can not be killed. These tasks behaves differently at shutdown. See the Architecture manual for details.
soft_task_def_ctrl_jet(m)	If called, the Kernel must store JET informations for the task.
soft_task_def_period(m,p)	Set the task period to p.
soft_task_def_met(m,d)	Set the task Mean Execution Time (MET) to d.
soft_task_def_wcet(m,w)	Set the Worst Case Execution Time to w.
soft_task_def_periodic(m)	Declare that the task is Periodic.
soft_task_def_aperiodic(m)	Declare that the task is Sporadic (Aperiodic).
soft_task_def_joinable(m)	Declare that the task is joinable with task_join()/pthread_join().
soft_task_def_unjoinable(m)	Declare that the task is not joinable (is detached) with task_join()/pthread_join().
soft_task_def_trace(m)	Declare that the task has to be traced by the Tracer.
soft_task_def_notrace(m)	Declare that the task has not to be traced by the Tracer.
soft_task_def_save_arrivals(m)	The task will save a pending activation if it arrives when the task is already active.
soft_task_def_skip_arrivals(m)	The task will ignore a pending activation if it arrives when the task is already active.

Table 1.2: SOFT_TASK_MODEL Macros

1.3 NRT_TASK_MODEL

```
typedef struct {
   TASK_MODEL t;
   int weight;
   TIME slice;
   int arrivals;
   int policy;
   int inherit;
   } NRT_TASK_MODEL;
```

A NRT task has a weight and a time slice, plus a policy attribute. It can be used to model Round Robin, Proportional Share, POSIX, and Priority tasks.

Note that policy and inherit are inserted in the model to support POSIX compliant scheduling without adding another Task Model; weight can be used to contain the fixed priority of a task; slice can be used to contain the RR slice of the task.

Table 1.3 shows the macros that applies to a `NRT_TASK_MODEL`.

1.4 JOB_TASK_MODEL

```
typedef struct {
   TASK_MODEL t;
   TIME period;
   struct timespec deadline;
   int noraiseexc;
   } JOB_TASK_MODEL;
```

This model implements a Job with an optional period and a starting deadline (for the first activation). A Job task can raise a XDEADLINE_MISS exception; if the flag noraiseexc is != 0, the exception is not raised.

This model is normally used with aperiodic servers: the aperiodic server insert a guest task in another level with that model calling `guest_create` and `guest_activate`. When the task has to be removed, `guest_end` is called.

Note that there is no capacity control on this model. Note that the task that accept this task DOESN'T reactivate the task after a period... There is NOT a guest_endcycle defined for this model...

Table 1.4 shows the macros that applies to a `JOB_TASK_MODEL`.

1.5 BDEDF_RES_MODEL (BlockDevice EDF resource model)

TBD

1.6 BDPSCAN_RES_MODEL (Block Device PSCAN resource model)

TBD

Macro name	Behaviour
nrt_task_default_model(m)	Default values for the Model (save arrivals, NRT_RR_POLICY, NRT_EXPLICIT_SCHED, others = 0).
nrt_task_def_level(m,l)	Set the Model level to l. A Module registered at level x can accept a Model only if it has level 0 or x. The default value is 0.
nrt_task_def_arg(m,a)	Set the void * argument passed to the task the first time it is activated. The default value is NULL.
nrt_task_def_stack(m,s)	Set the task stack size. The default value is 4096.
nrt_task_def_stackaddr(m,s)	If the stack is statically allocated you can tell its address here. The default is NULL (no pre-allocated stack).
nrt_task_def_group(m,g)	Set the task group to g. Task grouping influences primitives like group_activate() or group_kill()
nrt_task_def_usemath(m)	Declare that the task uses floating point arithmetic.
nrt_task_def_system(m)	Declare that the task is a system task. System tasks behaves differently at shutdown. See the Architecture manual for details.
nrt_task_def_nokill(m)	Declare that the task can not be killed. These tasks behaves differently at shutdown. See the Architecture manual for details.
nrt_task_def_ctrl_jet(m)	If called, the Kernel must store JET informations for the task.
nrt_task_def_weight(m,w)	Set the task weight to w.
nrt_task_def_slice(m,d)	Set the timeslice to d.
nrt_task_def_policy(m,p)	Set the policy of the task. p can be NRT_RR_POLICY or NRT_FIFO_POLICY. (used for POSIX scheduling)
nrt_task_def_inherit(m,i)	Tell if the task should inherit the same properties of the father. i can be NRT_INHERIT_SCHED or NRT_EXPLICIT_SCHED. (used for POSIX scheduling)
nrt_task_def_joinable(m)	Declare that the task is joinable with task_join()/pthread_join().
nrt_task_def_unjoinable(m)	Declare that the task is not joinable (is detached) with task_join()/pthread_join().
nrt_task_def_trace(m)	Declare that the task has to be traced by the Tracer.
nrt_task_def_notrace(m)	Declare that the task has not to be traced by the Tracer.
nrt_task_def_save_arrivals(m)	The task will save a pending activation if it arrives when the task is already active.
nrt_task_def_skip_arrivals(m)	The task will ignore a pending activation if it arrives when the task is already active.

Table 1.3: NRT_TASK_MODEL Macros

Macro name	Behaviour
job_task_default_model(m,dl)	Default values for the Model (period = 0, deadline = dl, noraiseexc = 0)
job_task_def_level(m,l)	Set the Model level to l. A Module registered at level x can accept a Model only if it has level 0 or x. The default value is 0.
job_task_def_arg(m,a)	Set the void * argument passed to the task the first time it is activated. The default value is NULL.
job_task_def_stack(m,s)	Set the task stack size. The default value is 4096.
job_task_def_stackaddr(m,s)	If the stack is statically allocated you can tell its address here. The default is NULL (no pre-allocated stack).
job_task_def_group(m,g)	Set the task group to g. Task grouping influences primitives like group_activate() or group_kill()
job_task_def_usemath(m)	Declare that the task uses floating point arithmetic.
job_task_def_system(m)	Declare that the task is a system task. System tasks behaves differently at shutdown. See the Architecture manual for details.
job_task_def_nokill(m)	Declare that the task can not be killed. These tasks behaves differently at shutdown. See the Architecture manual for details.
job_task_def_ctrl_jet(m)	If called, the Kernel must store JET informations for the task.
job_task_def_period(m,p)	Set the period to p.
job_task_def_deadline(m,dl)	Set the deadline to dl.
job_task_def_noexc(m)	Set noraiseexc = 1.
job_task_def_yesexc(m)	Set noraiseexc = 1.
job_task_def_joinable(m)	Declare that the task is joinable with task_join()/pthread_join().
job_task_def_unjoinable(m)	Declare that the task is not joinable (is detached) with task_join()/pthread_join().
job_task_def_trace(m)	Declare that the task has to be traced by the Tracer.
job_task_def_notrace(m)	Declare that the task has not to be traced by the Tracer.

Table 1.4: HARD_TASK_MODEL Macros

1.7 PC_RES_MODEL (Priority Ceiling Resource Model)

```
typedef struct {
   RES_MODEL r;
   DWORD priority;
   } PC_RES_MODEL;
```

This Resource Model signal to the Priority Ceiling (PC) Module that the task may use the PC protocol for some mutexes.

Note that the PC Module consider the tasks created without using this resource models to have priority = MAX_DWORD (the lowest).

The macro that can be used to setup the Resource Model are the following:

`PC_res_default_model(res, prio)`

Initializes a PC_RES_MODEL with a priority equal to prio.

`PC_res_def_level(res,l)`

Set the Model level to l (in a way similar to what happens to task models).

1.8 SRP_RES_MODEL (Stack Resource Policy resource model)

```
typedef struct {
   RES_MODEL r;
   DWORD preempt;
   } SRP_RES_MODEL;
```

This Resource Model signal to the Stack Resource Policy (PC) Module that the task will use the SRP protocol.

Note that the SRP Module does not influence the schedule of any task that did not pass a SRP_RES_MODEL at its creation. The SRP Module uses another resource model that is embedded into the mutex structure. See `kernel/modules/srp.c` for details.

The macro that can be used to setup the Resource Model are the following:

`SRP_res_default_model(res, pre)`

Initializes a SRP_RES_MODEL with a preemption level equal to prio.

`SRP_res_def_level(res,l)`

Set the Model level to l (in a way similar to what happens to task models).

Chapter 2

Mutex attributes

Every mutex attribute encode a particular mutex protocol, and one of them must be passed to the mutex_init() primitive to specify the protocol used by a particular mutex. The following paragraphs describe in detail the mutex attributes. Their definitions can be found in `include/kernel/model.h`. Examples of use of mutex attributes can be found later in this document.

2.1 PI_mutexattr_t (Priority Inheritance Mutex Attribute)

```
typedef mutexattr_t PI_mutexattr_t;
```

The Priority Ceiling Mutex Attribute.
You can initialize that attribute using the static initializer `PI_MUTEXATTR_INITIALIZER` or calling the macro
`PI_mutexattr_default(a)`
where a is the mutex attribute.

2.2 NPP_mutexattr_t (Non Preemptive Protocol Mutex Attribute)

```
typedef mutexattr_t NPP_mutexattr_t;
```

The Non Preemptive Protocol Mutex Attribute. You can initialize that attribute using the static initializer `NPP_MUTEXATTR_INITIALIZER` or calling the macro
`NPP_mutexattr_default(a)`
where a is the mutex attribute.

2.3 PC_mutexattr_t (Priority Ceiling Mutex Attribute)

```
typedef struct { mutexattr_t a; DWORD ceiling; } PC_mutexattr_t;
```

The Priority Ceiling Mutex Attribute.

You can initialize that attribute using the static initializer `PC_MUTEXATTR_INITIALIZER` or calling the macro

`PC_mutexattr_default(at,c)`

where at is the mutex attribute, and c is the ceiling of the mutex.

2.4 SRP_mutexattr_t (Stack Resource Policy Mutex Attribute)

```
typedef mutexattr_t SRP_mutexattr_t;
```

The Stack Resource Policy Mutex Attribute. You can initialize that attribute using the static initializer `SRP_MUTEXATTR_INITIALIZER` or calling the macro

`SRP_mutexattr_default(a)`

where at is the mutex attribute.

2.5 NOP_mutexattr_t (No Protocol Mutex Attribute)

```
typedef mutexattr_t NOP_mutexattr_t;
```

The No Protocol Mutex Attribute.

You can initialize that attribute using the static initializer `NOP_MUTEXATTR_INITIALIZER` or calling the macro

`NOP_mutexattr_default(a)`

where at is the mutex attribute.

2.6 NOPM_mutexattr_t (No Protocol Multiple lock Mutex Attribute)

typedef mutexattr_t NOPM_mutexattr_t;

The No Protocol Multiple lock Mutex Attribute. You can initialize that attribute using the static initializer `NOPM_MUTEXATTR_INITIALIZER` or calling the macro

`NOPM_mutexattr_default(a)`

where at is the mutex attribute.

Chapter 3

Scheduling algorithms

3.1 DUMMY

Task Models Accepted:

DUMMY_TASK_MODEL - This Model is used only at system startup to register the dummy task. It cannot be used by the other modules.

Description: This Module implements a `dummy` task. A dummy task is a task like all the other task that simply does an infinite loop, does nothing inside.

Why we need such a Module? Look at the Scheduling Module Architecture: when the Kernel needs to schedule a task, it asks to all the registered modules if they have a task to schedule. The hyphotesis is that there must be ALWAYS a task to schedule. The dummy Scheduling Modules is used to register a module that have always a task to schedule (the dummy task).

The dummy Module is not needed if you can ensure that there will always be a task to schedule.

Exceptions Raised:

XUNVALID_GUEST This level doesn't support guests. When a guest operation is called, the exception is raised.

XUNVALID_DUMMY_OP The dummy task can't be created, or activated, or another (strange) problem occurred.

Usage: Usually the Dummy Module is the LAST scheduling module registered in the function `__kernel_register_levels__`.

Just insert the following line into `__kernel_register_levels__` to register the Module:
`dummy_register_level();`

If you have a CPU where the HLT instruction works, you can use the define `__HLT_WORKS__`. If that `#define` is defined, the idle loop of the `dummy` task will use `HLT`, maybe saving power.

Files: `include/modules/dummy.h - kernel/modules/dummy.c`

Implementation hints: You can look at the code to learn how to create a new task at startup using a `RUNLEVEL_INIT` function.

3.2 INTDRIVE (Interrupt Server)

Theoretical Description by Giacomo Guidi <giacomo@gandalf.sssup.it>

3.2.1 Introduction

Trying to execute an IRQ and a timer handler, coming from a device driver, inside a task context, it is a priority for each Real-Time OS. If we design these drivers considering possible preemptions, exectution times and other Real-Time contraints, a schedulability test can guarantie our system.

Buf if we must reuse a source code from third-party drivers makers, as S.Ha.R.K. does, without having knowledge about the driver timing behaviour, with possible non-preemptable critical section, it is very difficult to impose contraints for a schedulability analisys.

Server are powerful tool for bounding sporadic requests, as IRQ and timers appear inside the OS. Anyway there are limitations considering these handlers as normal tasks inside a multitask environement. First of all, often you cannot make preemption on them.

The preemption is possible only with a previous analisys of the source code, assuring that the driver will not need a sequence of not-interruptible operation.

If we don't want to spend time to look inside a third-party driver, a solution can be to design a new server, which bounds the IRQ and timers requests, managing them as non-preemptable code.

3.2.2 Why do we need a new server technology ?

Due to the incredible amount of server algorithm, a doubt is raising: do we need another one ? There different factors which make the answer positive.

In the Fixed Priority world the Interrupt Handling is usually relegated to the highest priority thread. It means that no preemption can occours but also that the IRQ handlers and timers are executed immediately.

With unsafe non real-time drivers, it can be useful limits the bandwidth consumed by device handlers, executing them in a safe enviroment. A bandwidth limiting mechanism is achived with a server algorithm like the very well-known SS, DS and similar

But if we cannot make preemption, we cannot also stop the execution of an handler when the capacity is expired. If the capacity goes negative, we break the rules of the common server algorithm and, more important, we are not so sure about guarantes and performances.

Inside the Dynamic Priority scheduler like EDF and relative servers as CBS, this problem is harder. It is possible to emulate a priority mechanism using the shortest relative deadline for driver interrupts and timers. This is not complex with a set o periodic task, but quite impossible if a sporadic server is present, forcing the system designer to make an IRQ handler task as a non preemptable.

Anyway if we can guarantie our system also with a non preemptive task, the CBS and TBS algorithm don't consider the possibility to have a negative capacity, with unpredicatable performance change.

A solution for this problem could be modify an existing server technology to fit this possibility, but there is a simpler way and it is made possible through the hierarchical scheduling.

From a hierarchical point of view, the interrupt and timer server can be first level of scheduler. When this server don't execute, the idle time is used by a second scheduler, which is the main system scheduler.

Inside S.Ha.R.K. this is possible. The modules stack is instrinsically hierarchical and the idle time of the first kernel module becomes the execution time for the next one. Splitting the scheduler, our server algorithm can be system independent and it is possible to place it inside both FP and EDF applications.

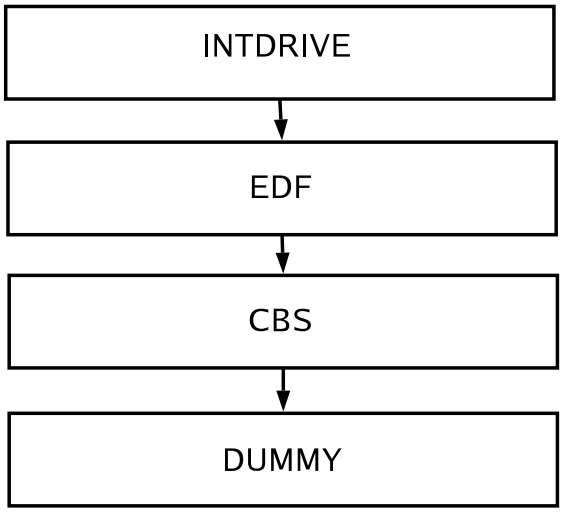

Figure 3.1: S.Ha.R.K. module stack

Through this approach we can also keep the server algorithm simple as possible, just to fit our requirements for the IRQ and timers handling, leaving the main scheduler out from this specific problem.

3.2.3 Server Postulates and Definition

The server postulates

- **IRQ and timer handlers are not preemptable** It must be requirement for a generic device driver
- **There is a maximum execution time for the handlers (HWCET)** Even if we don't know the exeact execution time of each handler we have to provide a worst case value to analize the system.
- **Handlers can be delayed by a maxium value (MDT)** The MDT is very close to a deadline and it represent the maximum stress condition for a device driver. Overcoming the MDT the handler are executed too late and the driver functionalities are not guarantie.

The server goals

- **To limit the handlers bandwidth to a maximum value** U**.** $U = \frac{Q}{T}$., where Q is the capacity the server provide each period T
- **To minimize the handler response time** After we are sure to not overcome the maximum capacity, the response time is the parameters to evaluate our server performance.
- **To make possible a hierarchical implementation** Hierarchical approach provides implementation flexibility on FP and EDF application. Moreover it allows a schedulability test for our system

A simple server idea which fit postulates and reach the goals. it is based on a three states automata. Looking at 3.2 we can specify the rules

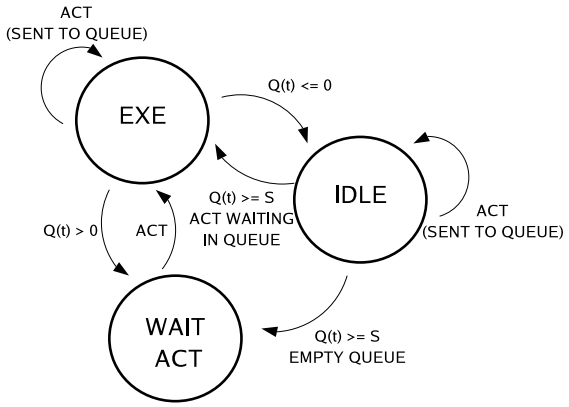

Figure 3.2: Three states server automata

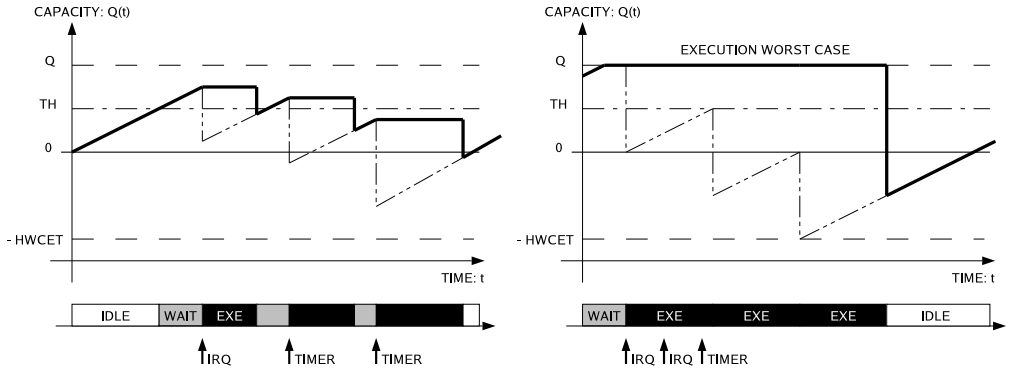

Figure 3.3: Examples of server capacity behaviour

- The server has a specific capacity Q and period T
- At the init condition the server is in [IDLE] with $Q(t) = 0$, where $Q(t)$ is the available capacity at time t.
- When an IRQ or timer is posted [ACT], if the server is in [WAIT ACT] and $Q(t) > 0$ the server executes the handler [EXE], else the event is sent to the activation queue
- When the server ends the handler execution, the used capacity is accounted to $Q(t)$. If $Q(t) <= 0$ the next state id [IDLE] else it switch to [WAIT ACT]
- The available capacity grows with a constant velocity equals to $\frac{Q}{T}$. The function $Q(t) = Q(t_l) + \frac{(t-t_l)Q}{T}$ returns the value of avail $Q(t)$ where $Q(t_l)$ is the last know value of $Q(t)$
- The server status switch from [IDLE] to [WAIT ACT] if $Q(t) > TH$, where TH is a threshold between 0 and Q. If the wait queue is not empty the server execute the first request.

3.2.4 Server Properties

- The medium used bandwidth is $BW_{mean} = B = \frac{Q}{T}$. The maximum used bandwidth will be $BW_{max} = B$ considering as minimum temporal window $T_{window} = (HWCET + \frac{Q}{1-B})\frac{T}{Q}$
- If the queue is empty, the maximum Response Time for an IRQ or timer request is $RT_{max} = (HWCET + TH)\frac{T}{Q}$
- The server idle time can be summarized as a two parameters set (α,δ), where $\alpha = 1 - \frac{Q}{T}$ and $\delta = HWCET + \frac{Q}{1-B}$. Using these two parameters a hierarchical analisys proposed by Bini is possible and the system can be guarantie.

3.2.5 Properties Demonstration

To maintain a constant bandwidth $B = \frac{Q}{T}$ if the server execute for a time T_{exec} must be idle for a time $T_{idle} = T_{exe} * T/Q - T_{exe}$. $B = \frac{T_{exec}}{T_{exe}+T_{idle}} = \frac{T_{exec}}{T_{exe}*T/Q-T_{exe}+T_{exe}} = \frac{Q}{T}$

The server uses at time t a capacity $Q_{exec}(t)$. This capacity can be split in two parts. If $Q_{exec}(t) > TH$ then $Q_1(t) = Q_{exec}(t) - TH$ and $Q_2(t) = TH$. If $Q_{exec}(t) <= TH$ then $Q_1(t) = 0$ and $Q_2(t) = Q_{exec}(t)$. For definition $Q(t) = Q_1(t) + Q_2(t)$.

To get $Q_1(t)$ inside the server, you need an idle time $I_1 = Q_1(t) * \frac{T}{Q}$ and this idle time is spent in the past ($t < 0$) because if you overcome the threshold you waited this at least that time. To get $Q_2(t)$ the idle time is $I_2 = Q_2(t) * \frac{T}{Q}$ and this time will be take in the future ($t > 0$), when the server will end its capacity. So, for every possible threshold and possible $Q_{exec}(t)$, the mean bandwidth will be $B = \frac{Q_1(t)+Q_2(t)}{Q_1(t)+Q_2(t)+I_1+I_2} = \frac{Q}{T}$. The maximum bandwidth implies the definition of a minimal temporal window. The worst case condition for our server is showed in figure . When a serie of consecutive activations brings the capacity $Q(t) = 0$ and HWCET request executes.

Task Models Accepted:

HARD_TASK_MODEL - Interrupt Hard Tasks. There is no period, the task can be only sporadic. The `wcet` must be != 0. This parameter will be used to set the Handler Worst Case Execution Time (HWCET).

NOTE: There is not ON-LINE guarantie !!! You need the Bini hierarchical approach to guarantie the system.

Usage: Usually this model is registered as one of the first Modules in the `__kernel_register_levels__` function. To register this module, just put this line into the `__kernel_register_levels__` function:

```
INTDRIVE_register_level(INTDRIVE_Q, INTDRIVE_T, INTDRIVE_FLAG);
```

INTDRIVE_Q: The server capacity

INTDRIVE_T: The server recharge time (look at the algorithm description)

INTDRIVE_FLAG:

(No_flags_enabled) - No checks

INTDRIVE_CHECK_WCET - When a wcet overrun occurs, an exception is raised.

Files: `include/modules/intdrive.h - kernel/modules/intdrive.c`

Implementation hints:

NOTES:

1. Inside the demos the IntDrive parameters are very high, you can lower them if you need to guarantie the system from the interrupts behaviour.

3.3 EDF (Earliest Deadline First)

Task Models Accepted:

HARD_TASK_MODEL - Hard Tasks (Periodic and Sporadic). The `wcet` and `mit` must be != 0. These parameters will be used to set the Worst Case Execution Time (WCET) and Period of the tasks. The `periodicity` can be either `PERIODIC` or `APERIODIC`. drel field must be <= mit.

NOTE: A Relative Deadline of 0 is interpreted as MIT.

NOTE: The utilization of the task is computed as wcet/drel.

NOTE: **offset** field specifies a release offset relative to task_activate or group_activate.

Guest Models Accepted:

JOB_TASK_MODEL - A single guest task activation. It must be identified by an absolute deadline and a period. The `period` field is ignored.

Description: This module schedules periodic and sporadic tasks based on their absolute deadlines. The task guarantee is based on a simple utilization approach. The utilization factor of a task is computed as wcet/drel. (By default, drel = mit.) A periodic task must only be activated once; subsequent activations are triggered by an internal timer. By contrast, an sporadic task must be explicitely activated for each instance. NO GUARANTEE is performed on guest tasks. The guarantee must be performed by the level that inserts guest tasks in the EDF level.

Exceptions raised:

XDEADLINE_MISS If a task misses its deadline and the `EDF_ENABLE_DL_EXCEPTION` flag is set, this exception is raised. Note that after raising that exception, the task can't be put in execution again. The safest thing to do is to Shut Down the system! This exception is also raised if a guest task miss its deadline.

XWCET_VIOLATION If a task executes longer than its declared wcet and the `EDF_ENABLE_WCET_EXCEPTION` flag is set, this exception is raised and the task is put in the `EDF_WCET_VIOLATED` state. To reactivate it, use `EDF_task_activate` via task_activate or manage directly the EDF data structure. Note that the exception is not handled properly, an `XDEADLINE_MISS` exception will also be raised at the period end.

XACTIVATION If a sporadic task is activated with a rate that is greather than the rate declared in the model, this exception is raised and the task is NOT activated. This exception is also raised if we are trying to activate a periodic task stopped with `task_delay`.

XUNVALID_GUEST This exception is raised if a `guest_endcycle` or `guest_sleep` guest calls are called.

Usage: Usually this model is registered as one of the first Modules in the `__kernel_register_levels__` function. To register this module, just put this line into the `__kernel_register_levels__` function:

`EDF_register_level(flag);`

where `flag` can be:

(No_flags_enabled) - Deadline and wcet overruns are ignored. Pending periodic jobs are queued and are eventually scheduled with correct deadlines according to their original arrival times. Sporadic tasks that arrive to often aresimply dropped.

EDF_ENABLE_DL_CHECK - When a deadline overrun occurs, the dl_miss counter of the task is increased. Same behavior for pending jobs as above.

EDF_ENABLE_WCET_CHECK - When a wcet overrun occurs, the wcet_miss counter of the task is increased. Same behavior for pending jobs as above.

EDF_ENABLE_DL_EXCEPTION - When a deadline overrun occurs, an exception is raised.

EDF_ENABLE_WCET_EXCEPTION - When a wcet overrun occurs, an exception is raised.

EDF_ENABLE_ACT_EXCEPTION When a periodic or sporadic task is activated too often, an exception is raised.

The functions `EDF_get_dl_miss`, `EDF_get_wcet_miss`, `EDF_get_act_miss`, and EDF_get_nact can be used to find out the number of missed deadlines, the number of wcet overruns, the number of skipped activations, and the number of currently queued periodic activations.

EDF_DISABLE_ALL - All checks disabled

EDF_ENABLE_GUARANTEE - Guarantee test enabled.

When enabled, an acceptance test ($\sum \frac{WCET_i}{Period_i} < 1$) is performed; Deadline miss exceptions are raised in any case.

EDF_ENABLE_ALL - All checks enabled

The EDF Module provides also addictional functions:

`bandwidth_t EDF_usedbandwidth(LEVEL l);`

It returns the used bandwidth, where l is the level at which the EDF Module is registered.

`int EDF_get_dl_miss(PID p);`

It returns the deadline miss counter.

`int EDF_get_wcet_miss(PID p);`

It returns the wcet miss counter.

`int EDF_get_act_miss(PID p);`

It returns the activation miss counter.

Files: `include/modules/edf.h - kernel/modules/edf.c`

Implementation hints: It uses the CONTROL_CAP field to keep track of the task execution time. It implements a ZOMBIE state for ended tasks. It post a single OSLib event for each task for deadline/reactivation/zombie detection. The `guest_endcycle` and `guest_sleep` guest calls are not implemented.

NOTES:

1. Relative deadlines **drel** <= **mit** may be specified.
2. An offset > 0 will delay the activation of the task by the same amount of time. To synchronize a group of tasks, assign suitable offsets and then use the group_activate function.
3. This level doesn't manage the main task.
4. The level uses the priority and timespec_priority fields.
5. The guest tasks don't provide the guest_endcycle function.

3.4 POSIX (fixed priority FIFO/RR scheduler)

Task Models Accepted:

NRT_TASK_MODEL - Non-Realtime Tasks. If the `inherit` field is set to `NRT_INHERIT_SCHED`, the scheduling properties of the running task are inherited (as required by the POSIX standard). Else, the `slice` field is used to set the time slice of the task (if `slice` is set to 0, the default value is used); the `weight` field is used to set the task priority; the `policy` field is used to set the policy of the task (`NRT_RR_POLICY` for Round-Robin or `NRT_FIFO_POLICY` for FIFO); the `arrivals` field can be set to `SAVE_ARRIVALS` or `SKIP_ARRIVALS`.

Description: This module schedule his tasks following the POSIX scheduler as described by the standard IEEE 1003.1c. This Module is typically used by the POSIX primitives for their scheduling pourposes. For example, the pthread_create function calls the task_createn function passing a NRT_TASK_MODEL filled with the correct parameters.

The Module can create the `__init__` task, that is typically used to call the Standard C `main()` function.

Exceptions Raised:

XUNVALID_GUEST This level doesn't support guests. When a guest operation is called, the exception is raised.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
POSIX_register_level(slice, createmain, mb, prioritylevels);
```

where `slice` is the default timeslice for the POSIX tasks scheduled using the Round-Robin policy, `createmain` is `POSIX_MAIN_YES` or `POSIX_MAIN_NO` if you want that the POSIX Module creates or not the `__init__` task, `mb` is the `struct multiboot_info *` the Kernel passed to the `__kernel_register_levels__` function, and prioritylevels are the number of different priority levels supported by the Module[1].

[1]The POSIX standard requires a minimum of 32 priority levels...

The POSIX Module provides also some addictional function, that are used to implement the behaviour of various POSIX primitives:

`int POSIX_sched_yield(LEVEL l);`

This function forces the running task to go to his queue tail, then calls the scheduler and changes the context. It is used by the `sched_yield()` primitive.

`int POSIX_get_priority_max(LEVEL l);`

This function returns the maximum level allowed for the POSIX level. It is used by the `sched_get_priority()` primitive.

`int POSIX_rr_get_interval(LEVEL l);`

This function returns the default timeslice for the POSIX level. It is used by the `sched_rr_get_interval()` primitive.

`int POSIX_getschedparam(LEVEL l, PID p, int *policy, int *priority);`

This functions returns some paramaters of a task; `policy` is set to `NRT_RR_POLICY` or `NRT_FIFO_POLICY`; priority is set in the range `[0..prioritylevels]`. The function returns `ENOSYS` or `ESRCH` if there are problems. The function must be called with interrupts disabled and is used by the `pthread_getschedparam()` primitive.

`int POSIX_setschedparam(LEVEL l, PID p, int policy, int priority);`

This functions sets some paramaters of a task; `policy` must be set to `NRT_RR_POLICY` or `NRT_FIFO_POLICY`;priority must be set in the range `[0..prioritylevels]`. The function returns `ENOSYS`, `EINVAL` or `ESRCH` if there are problems. The function must be called with interrupts disabled and is used by the `pthread_setschedparam() primitive`.

Files: `include/modules/posix.h - kernel/modules/posix.c`

Implementation hints: The implementation of this module is directly derived from the RR scheme.

3.5 RM (Rate Monotonic)

Task Models Accepted:

HARD_TASK_MODEL - Hard Tasks (Periodic and Sporadic). The `wcet` and `mit` must be != 0. These parameters will be used to set the Worst Case Execution Time (WCET) and Period of the tasks. The `periodicity` can be either `PERIODIC` or `APERIODIC`. The Relative Deadline is ignored (deadlines are equal to periods).

Guest Models Accepted:

JOB_TASK_MODEL - A single guest task activation. It must be identified by an absolute deadline and a period.

Description: This module schedule his tasks following the classic RM scheme as described by Liu and Layland. The task guarantee is based on the factor utilization approach. The tasks scheduled are periodic and sporadic. The sporadic tasks are like hard task with periodicity set to `APERIODIC`; they are guaranteed as a periodic task with period equal to the minimum interarrival time. All the task are put in a queue and the scheduling is based on the period value.

No Guarantee is performed on guest tasks. The guarantee must be performed by the level that inserts guest tasks in the RM level.

If a RM task does not respect its WCET and deadline constraints, then the Module will raise an exception. Note that the deadline exception is not recoverable, so the Module will be in an inconsistent state after a Deadline Miss. Deadline Miss in this Module are treated as unrecoverable errors.

If you try to activate a Periodic task that is not sleeping, nothing happens.

Exceptions Raised:

XDEADLINE_MISS If a task miss his deadline, the exception is raised. Note that after raising that exception, the task can't be put in execution again. The safest thing to do is to Shut Down the system! This exception is also raised if a guest task miss its deadline.

XWCET_VIOLATION If a task doesn't end the current cycle before if consume the `wcet`, an exception is raised, and the task is put in the `RM_WCET_VIOLATED` state. To reactivate it, use `RM_task_activate` via task_activate or manage directly the RM data structure. Note that the exception is not handled properly, an `XDEADLINE_MISS` exception will also be raised at the period end.

XACTIVATION If a sporadic task is activated with a rate that is greather than the rate declared in the model, this exception is raised and the task is NOT activated. This exception is also raised if we are trying to activate a periodic task stopped with `task_sleep` before the deadline in which the `task_sleep` is called.

XUNVALID_GUEST This exception is raised if a `guest_endcycle` or `guest_sleep` guest calls are called.

Usage: Usually this model is registered as one of the first Modules in the `__kernel_register_levels__` function. To register this module, just put this line into the `__kernel_register_levels__` function:

```
RM_register_level(flag);
```

where `flag` can be:

Restrictions & special features:

(No_flags_enabled) - Deadline and wcet overruns are ignored. Pending periodic jobs are queued and are eventually scheduled with correct deadlines according to their original arrival times. Sporadic tasks that arrive to often are simply dropped.

RM_ENABLE_DL_CHECK - When a deadline overrun occurs, the dl_miss counter of the task is increased. Same behavior for pending jobs as above.

RM_ENABLE_WCET_CHECK - When a wcet overrun occurs, the wcet_miss counter of the task is increased. Same behavior for pending jobs as above.

RM_ENABLE_DL_EXCEPTION - When a deadline overrun occurs, an exception is raised.

RM_ENABLE_WCET_EXCEPTION - When a wcet overrun occurs, an exception is raised.

RM_ENABLE_ACT_EXCEPTION When a periodic or sporadic task is activated too often, an exception is raised.

RM_DISABLE_ALL - Wcet and Guarantee test disabled

RM_ENABLE_GUARANTEE - Guarantee test enabled (when enabled, an acceptance test ($\sum \frac{WCET_i}{Period_i} < 1$) is performed; Deadline miss exceptions are raised in any case. Note that, for reasons of semplicity, the test that has been implemented IS NOT the test for RM, but only a simple acceptance test.

RM_ENABLE_ALL - Wcet and Guarantee test enabled

The RM Module provides also addictional functions:

`bandwidth_t RM_usedbandwidth(LEVEL l);`
It returns the used bandwidth, where l is the level at which the RM Module is registered.

`int RM_get_dl_miss(PID p);`
It returns the deadline miss counter.

`int RM_get_wcet_miss(PID p);`
It returns the wcet miss counter.

`int RM_get_act_miss(PID p);`
It returns the activation miss counter.

Files: `include/modules/rm.h - kernel/modules/rm.c`

Implementation hints: The implementation of this module is very similar to the implementation of the EDF Module.

NOTE

1. Relative deadlines drel <= mit may be specified.
2. An offset > 0 will delay the activation of the task by the same amount of time. To synchronize a group of tasks, assign suitable
3. offsets and then use the group_activate function.
4. This level doesn't manage the main task.
5. The level uses the priority and timespec_priority fields.
6. The guest tasks don't provide the guest_endcycle function.
7. At init time, the user can specify the behavior in case ofdeadline and wcet overruns. The following flags are available:

3.6 RR (Round Robin)

Task Models Accepted:

NRT_TASK_MODEL - Non-Realtime Tasks. The `slice` field is used to set the time slice of the task. If `slice` is set to 0, the default value is used. The `weight`, `arrivals`, `policy` and `inherit` fields are ignored.

Description: This module schedule his tasks following the classic round-robin scheme. The default timeslice is given at registration time and is a a per-task specification. The default timeslice is used if the `slice` field in the `NRT_TASK_MODEL` is 0.

If a task is activated when it is already active (its instance is not yet finished on a `task_endcycle` or `task_sleep`), nothing happens.

If you need to remember the pending activations you can use the `RR2` scheduling module.

The Module can create the `__init__` task, that is typically used to call the Standard C `main()` function.

Exceptions Raised:

XUNVALID_GUEST This level doesn't support guests. When a guest operation is called, the exception is raised.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
RR_register_level(slice, createmain, mb);
```

where `slice` is the default timeslice for the RR tasks, `createmain` is `RR_MAIN_YES` or `RR_MAIN_NO` if you want that RR creates or not the `__init__` task, `mb` is the `struct multiboot_info *` the Kernel passed to the `__kernel_register_levels__` function.

You can use more than one RR Scheduling Module to simulate a Multilevel scheduling policy. The RR Module does not add any addictional function.

Files: `include/modules/rr.h - kernel/modules/rr.c`

Implementation hints: This is one of the simplest scheduling module. It can be useful to learn how to create the __init__ task at startup time. It uses the CONTROL_CAP field. It supports negative task capacities (that can happen when using shadows; for that reason, the scheduler has a while inside, and the timeslices ar added and not assigned). No Guarantee is performed at task creation.

3.7 RR2 (Round Robin with pending activations)

Task Models Accepted:

NRT_TASK_MODEL - Non-Realtime Tasks. The `slice` field is used to set the time slice of the task. If `slice` is set to 0, the default value is used. The `arrivals` field is used to say if the activations have to be saved (`SAVE_ARRIVALS`) or skipped (`SKIP_ARRIVALS`). The `weight`, `policy` and `inherit` fields are ignored.

Description: The Module is identical to the RR Scheduling Module, except that task activations can be saved if a task was created with the arrivals field equal to SAVE_ARRIVALS.

Exceptions Raised: See the RR Scheduling Module.

Usage: See the RR Scheduling Module, changing every occurrence of RR with RR2.

Files: `include/modules/rr2.h - kernel/modules/rr2.c`

Implementation hints: With respect to the RR Scheduling Module, it adds a pending activation counter (nact) for each task.

3.8 RRSOFT (hard/SOFT Round Robin)

Task Models Accepted:

HARD_TASK_MODEL - Hard Tasks (Periodic and Sporadic). Only the periodicity and the mit parameters are used to know if the task is periodic/sporadic and to set the task period. The task timeslice is set to the default value.

SOFT_TASK_MODEL - Soft Tasks (Periodic and Sporadic). Only the periodicity, arrivals and period parameters are used to know if the task is periodic/sporadic, to set the task period and to know if the pending activations should be saved. The task timeslice is set to the default value.

NRT_TASK_MODEL - Non-Realtime Tasks. The `slice` field is used to set the time slice of the task. If `slice` is set to 0, the default value is used. The `arrivals` field is used to say if the activations have to be saved (`SAVE_ARRIVALS`) or skipped (`SKIP_ARRIVALS`). The `weight`, `policy` and `inherit` fields are ignored.

Description: This module can be used as a polymorphic module that can accept Hard, Soft or NRT Task Models. The policy used to schedule the tasks is the same of RR2, plus the fact that SOFT and HARD tasks can be periodic, so they can be automatically activated at each instance.

This Module is very useful if you want to replace another Module that accept Hard of Soft tasks with a round-robin scheduler, for example to compare a scheduling algorithm with the plain round robin.

Exceptions Raised:

XUNVALID_GUEST This level doesn't support guests. When a guest operation is called, the exception is raised.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
RRSOFT_register_level(slice, createmain, mb, models);
```

where `slice` is the default timeslice for the RR tasks, `createmain` is `RR_MAIN_YES` or `RR_MAIN_NO` if you want that RR creates or not the `__init__` task, `mb` is the `struct multiboot_info *` the Kernel passed to the `__kernel_register_levels__` function, and `models` specifies the kind of Models accepted by the Module. The `models` value can be an or of the values of the following constants: `RRSOFT_ONLY_HARD`, `RRSOFT_ONLY_HARD`, `RRSOFT_ONLY_HARD`.

You can use more than one RR Scheduling Module to simulate a Multilevel scheduling policy. The RR Module does not add any addictional function.

Files: `include/modules/rrsoft.h - kernel/modules/rrsoft.c`

Implementation hints: The implementation of the Module is similar to RR2 plus the implementation of the reactivation for periodic tasks.

Chapter 4

Aperiodic servers

4.1 CBS (Constant Bandwidth Server)

Task Models Accepted:

SOFT_TASK_MODEL - Soft Tasks (Periodic and Sporadic). The `met` and `period` must be != 0. These parameters will be used to set the Mean Execution Time (MET) and the Period of the tasks. The `periodicity` can be either `PERIODIC` or `APERIODIC`. The `arrivals` field can be either `SAVE_ARRIVALS` or `SKIP_ARRIVALS`. The `wcet` field is ignored (Worst case execution times are not used by CBS).

Description: This module schedule his tasks following the CBS algorithm. The task guarantee is based on the factor utilization approach. The tasks scheduled are periodic and sporadic. The sporadic tasks are like hard task with periodicity set to `APERIODIC`. All the task are put as guest task in the scheduling queue of another Module (typically that Module is an EDF Module).

A CBS server is attached to each task, with the parameters passed in the `SOFT_TASK_MODEL`. If you try to activate a Periodic task that is not sleeping, a pending activation is recorded[1].

Exceptions Raised:

XUNVALID_GUEST This level doesn't support guests. When a guest operation is called, the exception is raised.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
CBS_register_level(flag, master);
```

where `flag` can be:

CBS_DISABLE_ALL - Guarantee test disabled.

CBS_ENABLE_GUARANTEE - Guarantee test enabled (when enabled, an acceptance test ($\sum \frac{met_i}{period_i} < 1$) is performed; Deadline miss exceptions are raised in any case.

CBS_ENABLE_ALL - Guarantee test enabled.

[1] If the task was created with the `SAVE_ARRIVALS` option...

and `master` is the level to that the CBS is attached. At the moment, you can attach a CBS Module either to an `EDF` or an `EDFACT` Module. The CBS Module can be registered as the last scheduling Module after the `DUMMY` Module (this because the CBS Module does not use background time, and because when a CBS task is ready it is inserted in another queue!).

The CBS Module provides also an addictional function, that can be used to get the used bandwidth by the Module and the pending activations of a task. The prototypes of these function are:

`bandwidth_t CBS_usedbandwidth(LEVEL l);`

where l is the level at which the CBS Module is registered.

`int CBS_get_nact(LEVEL l, PID p);`

Returns the number of pending activations of a task. No control is done if the task is not a CBS task! (l is the level at which the CBS Module is registered, p is the PID of the task).

Files: `include/modules/cbs.h - kernel/modules/cbs.c`

Implementation hints: CBS implementation is similar to the EDF implementation except that a capacity exaustion postpones a deadline, and that the deadline event is simply a reactivation. This is one of the only Modules where the `task_eligible` task calls is defined non-void...

4.2 HARD_CBS

This Module is very similar to the standard CBS, but it implements the Hard Reservation algorithm.

Files: `include/modules/hardcbs.h - kernel/modules/hardcbs.c`

4.3 DS (Deferrable Server)

Task Models Accepted:

SOFT_TASK_MODEL - Soft Tasks (only Sporadic). The `periodicity` can be only `APERIODIC`. The `arrivals` field can be either `SAVE_ARRIVALS` or `SKIP_ARRIVALS`. The other fields are ignored.

Description: This module schedule its tasks following the Deferrable Server Algorithm. All the aperiodic requests are served on a FCFS basis. The Module can be configured to use only its budget or to also use the idle time left by the higher level Modules. The Module can be configured to save or skip task activations. The Module can be attached to either a RM or an EDF Module.

Exceptions Raised:

XUNVALID_GUEST This level doesn't support guests. When a guest operation is called, the exception is raised.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
DS_register_level(flag, master, Cs, per);
```

where `flag` can be:

DS_DISABLE_ALL - Guarantee test disabled; Background scheduling disabled.

DS_ENABLE_BACKGROUND - Background scheduling enabled.

DS_ENABLE_GUARANTEE_EDF - Guarantee test enabled (when enabled, an acceptance test ($\overline{U}_p + U_s < 1$) is performed. This flag have to be used if the Module is attached to an EDF Module.

DS_ENABLE_ALL_EDF - EDF guarantee test enabled, Background scheduling disabled.

DS_ENABLE_GUARANTEE_RM - Guarantee test enabled (when enabled, an acceptance test ($\overline{U}_p + U_s < ln\,2$) is performed. This flag have to be used if the Module is attached to a RM Module.

DS_ENABLE_ALL_RM - RM guarantee test enabled, Background scheduling disabled.

`master` is the level to that the DS Module is attached. At the moment, you can attach a DS Module either to an `EDF`, `EDFACT` or `RM` Module. `Cs` and `per` are the budget and the period of the deferrable server. If it is configured to do not use background time, the DS Module can be registered as the last scheduling Module after the `DUMMY` Module. Otherwise, it should be put in the middle of the list to catck the idle time left by higher level Modules.

The DS Module provides also an addictional function, that can be used to get the used bandwidth by the Module. The prototype of the function is:

```
bandwidth_t DS_usedbandwidth(LEVEL l);
```

where l is the level at which the DS Module is registered.

Files: `include/modules/ds.h - kernel/modules/ds.c`

Implementation hints: The DS Module uses a FIFO queue to enqueue the ready tasks. Only the first task is inserted into the queue of the Master Level. The Module does not use the CONTROL_CAP field, but it handles its capacity event by itself. The budget of the module is recharged every DS period, and is kept into an internal data structure of the module, and not in the wcet and avail_time fields of every task. Note that the idle time is recognized when the DS scheduler is called. In that case, no capacity event is posted. The implementation is very similar to the PS Module implementation.

4.4 PS (Polling Server)

Task Models Accepted:

SOFT_TASK_MODEL - Soft Tasks (only Sporadic). The `periodicity` can be only `APERIODIC`. The `arrivals` field can be either `SAVE_ARRIVALS` or `SKIP_ARRIVALS`. The other fields are ignored.

Description: This module schedule its tasks following the Polling Server Algorithm. All the aperiodic requests are served on a FCFS basis. The Module can be configured to use only its budget or to also use the idle time left by the higher level Modules. The Module can be configured to save or skip task activations. The Module can be attached to either a RM or an EDF Module.

Exceptions Raised:

XUNVALID_GUEST This level doesn't support guests. When a guest operation is called, the exception is raised.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
PS_register_level(flag, master, Cs, per);
```

where `flag` can be:

PS_DISABLE_ALL - Guarantee test disabled; Background scheduling disabled.

PS_ENABLE_BACKGROUND - Background scheduling enabled.

PS_ENABLE_GUARANTEE_EDF - Guarantee test enabled (when enabled, an acceptance test ($U_p + U_s < 1$) is performed. This flag have to be used if the Module is attached to an EDF Module.

PS_ENABLE_ALL_EDF - EDF guarantee test enabled, Background scheduling disabled.

PS_ENABLE_GUARANTEE_RM - Guarantee test enabled (when enabled, an acceptance test ($U_p + U_s < ln\,2$) is performed. This flag have to be used if the Module is attached to a RM Module.

PS_ENABLE_ALL_RM - RM guarantee test enabled, Background scheduling disabled.

`master` is the level to that the DS Module is attached. At the moment, you can attach a PS Module either to an `EDF`, `EDFACT` or `RM` Module. `Cs` and `per` are the budget and the period of the deferrable server. If it is configured to do not use background time, the PS Module can be registered as the last scheduling Module after the `DUMMY` Module. Otherwise, it should be put in the middle of the list to catck the idle time left by higher level Modules.

The PS Module provides also an addictional function, that can be used to get the used bandwidth by the Module. The prototype of the function is:

```
bandwidth_t PS_usedbandwidth(LEVEL l);
```

where l is the level at which the PS Module is registered.

Files: `include/modules/ps.h - kernel/modules/ps.c`

Implementation hints: The PS Module uses a FIFO queue to enqueue the ready tasks. Only the first task is inserted into the queue of the Master Level. The Module does not use the CONTROL_CAP field, but it handles its capacity event by itself. The budget of the module is recharged every PS period, and is kept into an internal data structure of the module, and not in the wcet and avail_time fields of every task. Note that the idle time is recognized when the PS scheduler is called. In that case, no capacity event is posted. The implementation is very similar to the DS Module implementation.

4.5 SS (Sporadic Server)

Task Models Accepted:

SOFT_TASK_MODEL - Soft Tasks (only Sporadic). The `periodicity` can be only `APERIODIC`. The `arrivals` field can be either `SAVE_ARRIVALS` or `SKIP_ARRIVALS`. The other fields are ignored.

Description: This module schedule its tasks following the Sporadic Server Algorithm. The Module can be configured to save or skip task activations. The Module can be attached to either a RM or an EDF Module. The module has been written by Marco Gigante. Please contact the author for more informations.

Exceptions Raised:

XUNVALID_GUEST This level doesn't support guests. When a guest operation is called, the exception is raised.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
SS_register_level(flag, master, Cs, per);
```

where `flag` can be:

SS_DISABLE_ALL - Guarantee test disabled; Background scheduling disabled.

SS_ENABLE_BACKGROUND - Background scheduling enabled.

SS_ENABLE_GUARANTEE_EDF - Guarantee test enabled (when enabled, an acceptance test ($U_p + U_s < 1$) is performed. This flag have to be used if the Module is attached to an EDF Module.

SS_ENABLE_ALL_EDF - EDF guarantee test enabled, Background scheduling disabled.

SS_ENABLE_GUARANTEE_RM - Guarantee test enabled (when enabled, an acceptance test ($U_p + U_s < ln\,2$) is performed. This flag have to be used if the Module is attached to a RM Module.

SS_ENABLE_ALL_RM - RM guarantee test enabled, Background scheduling disabled.

`master` is the level to that the SS Module is attached. At the moment, you can attach a SS Module either to an `EDF`, `EDFACT` or `RM` Module. `Cs` and `per` are the budget and the period of the deferrable server. If it is configured to do not use background time, the SS Module can be registered as the last scheduling Module after the `DUMMY` Module. Otherwise, it should be put in the middle of the list to catck the idle time left by higher level Modules.

The SS Module provides also some addictional function, that can be used to get the used bandwidth by the Module and its available capacity. The prototypes of the functions are:

```
bandwidth_t SS_usedbandwidth(LEVEL l);
```

where l is the level at which the SS Module is registered.

```
int SS_availCs(LEVEL l);
```

Returns tha available capacity of the Module

Files: `include/modules/ss.h - include/modules/ssutils.h - kernel/modules/ss.c`

Implementation hints: The implementation of the Module is quite complex. Please refer to the comments in the source code.

4.6 TBS (Total Bandwidth Server)

Task Models Accepted:

SOFT_TASK_MODEL - Soft Tasks (only Sporadic). The `wcet` must be != 0. The `periodicity` can be either `PERIODIC` or `APERIODIC`. The `arrivals` field can be either `SAVE_ARRIVALS` or `SKIP_ARRIVALS`. The other fields are ignored.

Description: This module schedule his tasks following the TBS algorithm. The task guarantee is based on the factor utilization approach. The tasks scheduled are only sporadic. Each task has a deadline assigned with the TBS scheme, $d_k = \max(r_k, d_{k-1}) + \frac{wcet}{U_s}$

The tasks are inserted in an EDF level (or similar), and the TBS level expects that the task is scheduled with the absolute deadline passed in the guest model.

The acceptance test is based on the factor utilization approach. The theory guarantees that the task set is schedulable if $U_p + U_s \leq 1$ so it is sufficient to add the Us to the bandwidth used by the upper levels.

Exceptions Raised:

XUNVALID_GUEST This level doesn't support guests. When a guest operation is called, the exception is raised.

XDEADLINE_MISS If a task miss his deadline, the exception is raised. Normally, a TBS task can't cause the raise of such exception because if it really use more time than declared a `XWCET_VIOLATION` is raised instead.

XWCET_VIOLATION If a task doesn't end the current cycle before if consume the wcet, an exception is raised, and the task is put in the `TBS_WCET_VIOLATED` state. To reactivate it, call `TBS_task_activate` using `task_activate` or manage directly the TBS data structure. Note that if the exception is not handled properly, an `XDEADLINE_MISS` exception will also be raised at the absolute deadline...

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
TBS_register_level(flag, master, num, den);
```

where `flag` can be:

TBS_DISABLE_ALL - Guarantee test disabled.

TBS_ENABLE_GUARANTEE - Guarantee test enabled (when enabled, the acceptance test is performed); Deadline miss exceptions are raised in any case.

TBS_ENABLE_WCET_CHECK - WCET check enabled.

TBS_ENABLE_ALL - Guarantee test and WCET check enabled.

and `master` is the level to that the TBS is attached. num and den are the reserved bandwidth ($U_s = \frac{num}{den}$) for the TBS server. At the moment, you can attach a TBS Module either to an `EDF` or an `EDFACT` Module. The TBS Module can be registered as the last scheduling Module after the `DUMMY` Module (this because the TBS Module does not use background time, and because when a TBS task is ready it is inserted in another queue!).

The TBS Module provides also some addictional functions, that can be used to get the used bandwidth by the Module and the pending activations of a task. The prototypes of these function are:

```
bandwidth_t TBS_usedbandwidth(LEVEL l);
```

where l is the level at which the TBS Module is registered.

```
int TBS_get_nact(LEVEL l, PID p);
```

Returns the number of pending activations of a task. No control is done if the task is not a TBS task! (l is the level at which the TBS Module is registered, p is the PID of the task).

Files: `include/modules/tbs.h - kernel/modules/tbs.c`

Implementation hints: The TBS implementation uses a FIFO queue for serving the aperiodic requests of the Module's tasks. Only the first task is put in the ready queue of the master module.

Chapter 5

Sharing resource access protocols

5.1 CABS

Description: This module implements the Cyclical Asynchronous Buffers as described in the Volume I of the User Manual.

Note that the Maximum number of CABs that can be created is MAX_CAB (defined in include/modules/cabs.h).

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
CABS_register_module();
```

Files: `include/modules/cabs.h - kernel/modules/cabs.c`

5.2 HARTPORT

Description: This module implements the Communication Ports as described in the Volume I of the User Manual.

Usage: To register this module, just put this line into the `__init__` task[1]:

```
HARTPORT_init();
```

IMPORTANT: Remember that to get the ports running, you need to register also the SEM module!

Files: `include/modules/hartport.h - kernel/modules/hartport.c- kernel/modules/hartport.his`

5.3 NOP (NO Protocol)

Resource Models Accepted: None.

[1] or in any other task; in any case this function has to be called before using the port primitives

Description: The NOP Module implements a binary semaphore using the mutex interface. If a task owns a mutex, it can not lock that mutex again until it has unlocked it. This protocol can produce priority inversions and chained blocking.

Exceptions Raised: None

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

`NOP_register_module();`

The NOP protocol uses the default mutex mechanism provided by the Kernel. In that way, the NOP behaviour for a mutex can be set at run-time, and the applications can choose the protocol of their mutexes without changing the task source code.

To initialize a NOP mutex, you need to call `mutex_init()` passing a previously initialized `NOP_mutexattr_t`.

In the following example, the demo task uses two resources, labeled with `m1` and `m2`, and accesses three different critical sections:

```
#include <modules/nop.h>
...
mutex_t m1,m2;
...
void *demo(void *arg)
{
  ...
  mutex_lock(&m1);
  /* critical section of m1 */
  ...
  mutex_unlock(&m1);
  ...
  mutex_lock(&m1);
  /* only m1 locked */
  ...
  mutex_lock(&m2);
  /* m1 and m2 locked */
  ...
  mutex_unlock(&m2); /* NOTE: first m2, then m1!  */
  mutex_unlock(&m1);
  ...
  return 0;
}
...
int main(int argc, char **argv)
{
  HARD_TASK_MODEL m;
  PID p0, p1, p2;

  NOP_mutexattr_t a;

  /* Initialize the NOP mutexes */
  NOP_mutexattr_default(a);
  mutex_init(&m1,&a);
  mutex_init(&m2,&a);

  ...

  /* Create the task */
  hard_task_default_model(m);
```

```
  hard_task_def_mit(m, 1000000);
  hard_task_def_wcet(m, 80000);
  p0 = task_create("DEMO", demo, &m, NULL);
  ...
}
```

Critical sections must be properly nested (like Chinese boxes): hence the order of the `mutex_unlock(m1)` and `mutex_unlock(m2)` primitives cannot be changed without modifying the corresponding `mutex_lock()`.

Note: NOP Mutexes can be statically allocated. See `include/modules/nop.h` for details.

Note: A task can use NOP mutexes with other mutexes with different protocol (for example, PI, PC, SRP mutexes). We don't know the behaviour of that choice, but, if you want to try, it works!

Files: `include/modules/nop.h - kernel/modules/nop.c`

Implementation hints: The implementation of the NOP mutexes is similar to the internal semaphores.

5.4 NOPM (NO Protocol Multiple lock)

Resource Models Accepted: None.

Description: The NOP Module implements a binary semaphore using the mutex interface. If a task owns a mutex, it can lock that mutex again. It is like the NOP Module, but the owner of the mutex can issue multiple lock/unlock on mutex. This protocol can produce priority inversions and chained blocking.

Exceptions Raised: None

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

`NOPM_register_module();`

The NOPM protocol uses the default mutex mechanism provided by the Kernel. In that way, the NOPM behaviour for a mutex can be set at run-time, and the applications can choose the protocol of their mutexes without changing the task source code.

To initialize a NOPM mutex, you need to call `mutex_init()` passing a previously initialized `NOPM_mutexattr_t`.

In the following example, the demo task uses two resources, labeled with `m1` and `m2`, and accesses three different critical sections:

```
#include <modules/nopm.h>
...
mutex_t m1,m2;
...
void *demo(void *arg)
{
  ...
```

```
  mutex_lock(&m1);
  /* critical section of m1 */
  ...
  mutex_unlock(&m1);
  ...
  mutex_lock(&m1);
  /* only m1 locked */
  ...
  mutex_lock(&m2);
  /* m1 and m2 locked */
  ...
  mutex_unlock(&m2); /* NOTE: first m2, then m1!  */
  mutex_unlock(&m1);
  ...
  return 0;
}
...
int main(int argc, char **argv)
{
  HARD_TASK_MODEL m;
  PID p0, p1, p2;

  NOPM_mutexattr_t a;

  /* Initialize the NOP mutexes */
  NOPM_mutexattr_default(a);
  mutex_init(&m1,&a);
  mutex_init(&m2,&a);

  ...

  /* Create the task */
  hard_task_default_model(m);
  hard_task_def_mit(m, 1000000);
  hard_task_def_wcet(m, 80000);
  p0 = task_create("DEMO", demo, &m, NULL);
  ...
}
```

Critical sections must be properly nested (like Chinese boxes): hence the order of the `mutex_unlock(m1)` and `mutex_unlock(m2)` primitives cannot be changed without modifying the corresponding `mutex_lock()`.

Note: A task can use NOPM mutexes with other mutexes with different protocol (for example, PI, PC, SRP mutexes). We don't know the behaviour of that choice, but, if you want to try, it works!

Files: `include/modules/nopm.h - kernel/modules/nopm.c`

Implementation hints: The implementation of the NOPM mutexes is similar to the NOP implementation, except that it counts the numer of times a task locked a mutex.

5.5 NPP (Non Preemptive Protocol)

Resource Models Accepted: None.

Description: The NPP Module implements a binary semaphore using the mutex interface. The difference with the NOP Module is that when a task lock a NPP mutex, it became non-preemptive as it called the `task_nopreempt()` primitive. Critical section can be nested.

Exceptions Raised:

XMUTEX_OWNER_KILLED This exception is raised when a task ends and it owns one or more mutexes.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

`NPP_register_module();`

The NPP protocol uses the default mutex mechanism provided by the Kernel. In that way, the NPP behaviour for a mutex can be set at run-time, and the applications can choose the protocol of their mutexes without changing the task source code.

To initialize a NPP mutex, you need to call `mutex_init()` passing a previously initialized `NPP_mutexattr_t`.

In the following example, the demo task uses two resources, labeled with `m1` and `m2`, and accesses three different critical sections:

```
#include <modules/npp.h>
...
mutex_t m1,m2;
...
void *demo(void *arg)
{
  ...
  mutex_lock(&m1);
  /* critical section of m1 */
  ...
  mutex_unlock(&m1);
  ...
  mutex_lock(&m1);
  /* only m1 locked */
  ...
  mutex_lock(&m2);
  /* m1 and m2 locked */
  ...
  mutex_unlock(&m2); /* NOTE: first m2, then m1!  */
  mutex_unlock(&m1);
  ...
  return 0;
}
...
int main(int argc, char **argv)
{
  HARD_TASK_MODEL m;
  PID p0, p1, p2;

  NPP_mutexattr_t a;

  /* Initialize the NOP mutexes */
  NPP_mutexattr_default(a);
  mutex_init(&m1,&a);
  mutex_init(&m2,&a);

  ...
```

```
  /* Create the task */
  hard_task_default_model(m);
  hard_task_def_mit(m, 1000000);
  hard_task_def_wcet(m, 80000);
  p0 = task_create("DEM0", demo, &m, NULL);
  ...
}
```

Critical sections must be properly nested (like Chinese boxes): hence the order of the `mutex_unlock(m1)` and `mutex_unlock(m2)` primitives cannot be changed without modifying the corresponding `mutex_lock()`.

Note: A task can use NPP mutexes with other mutexes with different protocol (for example, PI, PC, SRP mutexes; except for nested critical sections). We don't know the behaviour of that choice, but, if you want to try, it works!

Files: `include/modules/npp.h - kernel/modules/npp.c`

Implementation hints: The implementation of the NPP Module counts the number of locks a task issued and uses `task_preempt()`/`task_nopreempt()` to implement the protocol.

5.6 PC (Priority Ceiling)

Resource Models Accepted:

PC_RES_MODEL - Task priority. This model have to be used to tell to the PC Module that a task have to be scheduled following the PC rules. The model can be passed at creation time to more than one task, and it only contains the priority information.

Description: This Module implements the Priority Ceiling (PC) Protocol. This mechanism can be easily applied to Fixed Priority Scheduling to bound blocking times and to eliminate chained blocking and deadlocks.

Exceptions Raised:

XMUTEX_OWNER_KILLED This exception is raised when a task ends and it owns one or more PC mutexes.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

`PC_register_module();`

The PC protocol uses the default mutex mechanism provided by the Kernel. In that way, the PC behaviour for a mutex can be set at run-time, and the applications can choose the protocol of their mutexes without changing the task source code.

To initialize a PC mutex, you need to call `mutex_init()` passing a previously initialized `PC_mutexattr_t`, that contains the ceiling of the mutex.

To apply the Priority Ceiling protocol to a task, you need to tell the Module what is the priority of that task[2]. That information is given ti the Module using a Resource Model of type `PC_RES_MODEL` at task creation time (in other words, to the `task_createn` primitive). That model have to be previously initialized with the priority of the task.

Note that the priority value used by the PC Module may differ from the priority of the scheduling algorithm that really schedule the tasks. Typically, priorities are set to the same (or equipollent) values.

In the following example, the demo task uses two resources, labeled with `m1` and `m2`, and accesses three different critical sections:

```
#include <modules/pc.h>
...
mutex_t m1,m2;
...
void *demo(void *arg)
{
  ...
  mutex_lock(&m1);
  /* critical section of m1 */
  ...
  mutex_unlock(&m1);
  ...
  mutex_lock(&m1);
  /* only m1 locked */
  ...
  mutex_lock(&m2);
  /* m1 and m2 locked */
  ...
  mutex_unlock(&m2); /* NOTE: first m2, then m1!  */
  mutex_unlock(&m1);
  ...
  return 0;
}
...
int main(int argc, char **argv)
{
  HARD_TASK_MODEL m;
  PID p0, p1, p2;

  PC_mutexattr_t a;
  PC_RES_MODEL r;

  /* Initialize the SRP mutexes */
  PC_mutexattr_default(a,1); /* 1 is the pr.  level */
  mutex_init(&m1,&a);
  mutex_init(&m2,&a);

  ...

  /* Create the task */
  hard_task_default_model(m);
  hard_task_def_mit(m, 1000000);
  hard_task_def_wcet(m, 80000);

  PC_res_default_model(r, 1);     /* set the task Priority */
```

[2]Also a task that have to be scheduled using the PC rules but without using any PC mutex must declare its priority!

```
  p0 = task_create("DEMO", demo, &m, &r);
  ...
}
```

Critical sections must be properly nested (like Chinese boxes): hence the order of the `mutex_unlock(m1)` and `mutex_unlock(m2)` primitives cannot be changed without modifying the corresponding `mutex_lock()`.

The Module also provides three functions that are used to implement similar POSIX functions. Typically the user does not need to call them.

`int PC_get_mutex_ceiling(const mutex_t *mutex, DWORD *ceiling);`

This function gets the ceiling of a PC mutex, and it have to be called only by a task that owns the mutex. Returns -1 if the mutex is not a PC mutex, 0 otherwise.

`int PC_set_mutex_ceiling(mutex_t *mutex, DWORD ceiling, DWORD *old_ceiling);`

This function sets the ceiling of a PC mutex, and it have to be called only by a task that owns the mutex. Returns -1 if the mutex is not a PC mutex, 0 otherwise.

`void PC_set_task_ceiling(RLEVEL r, PID p, DWORD priority);`

This function sets the priority of a task.

Note: The Real-Time Literature typically found a feasibility test for the scheduling only in the case that all hard tasks use the PC protocol. Note that the Module is written in a way that only the tasks that declared their Priority will use the PC protocol. If a task that does not have a Priority, it is not scheduled using the PC protocol, also if it is inserted in the same scheduling queue of PC tasks.

Note: A task can use PC mutexes with other mutexes with different protocol (for example, PI, SRP, NOP mutexes). We don't know the behaviour of that choice, but, if you want to try, it works!

Files: `include/modules/pc.h - kernel/modules/pc.c`

Implementation hints: The PC Module uses the shadow mechanism to implement its behaviour. It keeps track of all the locked mutexes to block (setting the shadow field to the mutex owner) a task that requires a lock on a mutex with ceiling less than the system ceiling. When unlocking, all the shadows are reset. At that point they will try again to reaquire the mutex, that time starting from the highest priority task in the scheduling queue.

5.7 PI (Priority Inheritance)

Resource Models Accepted: None.

Description: This Module implements the Priority Inheritance mechanism. This mechanism can be easily applied to Fixed Priority Scheduling to bound blocking times.

Exceptions Raised:

XMUTEX_OWNER_KILLED This exception is raised when a task ends and it owns one or more PI mutexes.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

`PI_register_module();`

The PI protocol uses the default mutex mechanism provided by the Kernel. In that way, the PI behaviour for a mutex can be set at run-time, and the applications can choose the protocol of their mutexes without changing the task source code.

To initialize a PI mutex, you need to call `mutex_init()` passing a previously initialized `PI_mutexattr_t`.

In the following example, the demo task uses two resources, labeled with `m1` and `m2`, and accesses three different critical sections:

```
#include <modules/pi.h>
...
mutex_t m1,m2;
...
void *demo(void *arg)
{
  ...
  mutex_lock(&m1);
  /* critical section of m1 */
  ...
  mutex_unlock(&m1);
  ...
  mutex_lock(&m1);
  /* only m1 locked */
  ...
  mutex_lock(&m2);
  /* m1 and m2 locked */
  ...
  mutex_unlock(&m2); /* NOTE: first m2, then m1!  */
  mutex_unlock(&m1);
  ...
  return 0;
}
...
int main(int argc, char **argv)
{
  HARD_TASK_MODEL m;
  PID p0, p1, p2;

  PI_mutexattr_t a;

  /* Initialize the SRP mutexes */
  PI_mutexattr_default(a);
  mutex_init(&m1,&a);
  mutex_init(&m2,&a);

  ...

  /* Create the task */
  hard_task_default_model(m);
  hard_task_def_mit(m, 1000000);
  hard_task_def_wcet(m, 80000);

  p0 = task_createn("DEMO", demo, &m, NULL);
  ...
}
```

Critical sections must be properly nested (like Chinese boxes): hence the order of the `mutex_unlock(m1)` and `mutex_unlock(m2)` primitives cannot be changed without modifying the corresponding `mutex_lock()`.

Note: A task can use PI mutexes with other mutexes with different protocol (for example, SRP, PC, NOP mutexes). We don't know the behaviour of that choice, but, if you want to try, it works!

Files: `include/modules/pi.h - kernel/modules/pi.c`

Implementation hints: This is the simplest protocol that can be implemented using the shadows mechanism. The implementation set the shadow field when a task blocks, and reset it when the mutex is unlocked. At that point, the highest priority free task will lock the mutex.

5.8 SEM (POSIX Semaphores)

Description: This module implements the POSIX Semaphores as described in the Volume I of the User Manual.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
SEM_register_module();
```

Files: `include/modules/sem.h - kernel/modules/sem.c`

Implementation hints: The implementation uses a fixed number of semaphores (note that Internal Semaphores does not!). Note the use of `task_testcancel()` to implement the cancellation points.

5.9 SRP (Stack Resource Policy)

Resource Models Accepted:

SRP_RES_MODEL - Preemption levels. This model have to be used to tell to the SRP Module that a task have to be scheduled following the SRP rules. The model can be passed at creation time to more than one task, and it only contains the preemption level information.

Description: *Stack Resource Policy* (SRP) is a mutual exclusion mechanism suitable for hard real-time tasks, that can be used with static or dynamic scheduling algorithms. Its main feature is the property that a task is blocked not when it tries to use the resource, but when it tries to preempt another task owning the shared resource. This early blocking simplifies the protocol implementation and reduces the number of context switches, increasing the system efficiency.

This SRP implementation provides a simplified version of the protocol, suitable only for single-unit resources, that defines a particular behaviour for the `mutex_XXX` primitives. Moreover, it is

implemented using the shadows mechanism (described in the S.Ha.R.K. architecture manual), so there is no restriction on the Task Models of the tasks that will use SRP[3].

SRP is implemented associating a dynamic priority $p(\tau)$ and a preemption level $\pi(\tau)$ to each task τ. If an EDF scheduler is used to schedule the tasks that use SRP, we can say that the priority $p(\tau)$ of task τ is proportional to $1/d$, where d is the task's absolute deadline, while the preemption level $\pi(\tau)$ is proportional to $1/D$, where D is the tasks's relative deadline.

Each resource is assigned a dynamic *priority ceiling* C_R, defined as:

$$C_R = max\{0\},\ \{\pi(\tau) : R \in R(\tau)\, and\, Rsbusy\}$$

where $R(\tau)$ is the set of all resources used by task τ. According to this definition, if a resource is free, its ceiling is 0; if a resource is busy, its ceiling is the maximum among the preemption levels of tasks using it. A *System Ceiling* Π_S is also defined as the maximum ceiling among all the resources:

$$\Pi_S \ = \ max_r\{C_r \ : \ r = 1, \ldots, m\}.$$

Starting from these definitions, the SRP works according to the following rule:

> *The execution of a task is delayed until it becomes the task having the highest priority and its preemption level is strictly greater than the system ceiling.*

Two tests are needed to allow a task to execute: a test on its priority (i.e., the deadline, which is dynamically assigned) and a test on the preemption level (which is statically determined).

The SRP protocol ensures that a task τ, once started, cannot be blocked on any resource until it terminates. Consequently, SRP avoids unbounded priority inversion, blocking chains, and deadlocks. It also reduces the number of context switches and simplifies the implementation of the mechanism.

Since the SRP may be applied on various Scheduling Modules with different scheduling policies, no guarantee or blocking time calculations are performed by the Kernel.

The tasks that use SRP mutexes must declare the use of a shared resource, using the provided resource modules.

Finally, note that this implementation of SRP is NOT compatible with the join primitive. If a task that uses SRP mutexes calls `task_join` or `pthread_join`, the result is undefined.

Exceptions Raised:

XMUTEX_OWNER_KILLED This exception is raised when a task ends and it owns one or more SRP mutexes.

XSRP_UNVALID_LOCK This exception is raised when a task try to lock a srp mutex but it don't have the privilege.

Usage: To register this module, just put this line into the `__kernel_register_levels__` function:

```
SRP_register_module();
```

The SRP protocol uses the default mutex mechanism provided by the Kernel. In that way, the SRP behaviour for a mutex can be set at run-time, and the applications can choose the protocol of their mutexes without changing the task source code.

To initialize a SRP mutex, you need to call `mutex_init()` passing a previously initialized `SRP_mutexattr_t`.

To use a SRP mutex, you need to tell the Module two kind of informations:

[3]However, note that a schedulability test is only available for EDF and RM schedulers.

- First, every task that have to be scheduled following the SRP rules must have a (fixed) preemption level[4].
- Second, every task must declare the mutexes that it will use during its life.

All the two informations have to be given to the Module using Resource Models.

To set a task preemption level you need to pass an `SRP_RES_MODEL` at task creation time (in other words, to the `task_createn` primitive). That model have to be previously initialized with the preemption level of the task.

To declare that a task will use a particular mutex, you have to pass another Resource Model at task creation time. The Model you need to pass is returned by the following function:

`RES_MODEL *SRP_usemutex(mutex_t *m);`

where `m` is the SRP mutex the task will use.

In the following example, the demo task uses two resources, labeled with `m1` and `m2`, and accesses three different critical sections:

```
#include <modules/srp.h>
...
mutex_t m1,m2;
...
void *demo(void *arg)
{
  ...
  mutex_lock(&m1);
  /* critical section of m1 */
  ...
  mutex_unlock(&m1);
  ...
  mutex_lock(&m1);
  /* only m1 locked */
  ...
  mutex_lock(&m2);
  /* m1 and m2 locked */
  ...
  mutex_unlock(&m2); /* NOTE: first m2, then m1!  */
  mutex_unlock(&m1);
  ...
  return 0;
}
...
int main(int argc, char **argv)
{
  HARD_TASK_MODEL m;
  PID p0, p1, p2;

  SRP_mutexattr_t a;
  SRP_RES_MODEL r;

  /* Initialize the SRP mutexes */
  SRP_mutexattr_default(a);
  mutex_init(&m1,&a);
  mutex_init(&m2,&a);

  ...

  /* Create the task */
```

[4]Also a task that have to be scheduled using the SRP rules but without using any SRP mutex must declare its preemption level!

```
  hard_task_default_model(m);
  hard_task_def_mit(m, 1000000);
  hard_task_def_wcet(m, 80000);

  SRP_res_default_model(r, 3);     /* set the task Preemption level */

  p0 = task_createn("DEMO", demo, &m, &r,
          SRP_usemutex(&m1), SRP_usemutex(&m2), NULL);
  ...
}
```

Critical sections must be properly nested (like Chinese boxes): hence the order of the `mutex_unlock(m1)` and `mutex_unlock(m2)` primitives cannot be changed without modifying the corresponding `mutex_lock()`.

Note: The Real-Time Literature typically found a feasibility test for the scheduling only in the case that all hard tasks use the SRP protocol. Note that the Module is written in a way that only the tasks that declared their Preemption Level will use the SRP protocol. If a task that does not have a Preemption Level[5], it is not scheduled using the SRP protocol, also if it is inserted in the same scheduling queue of SRP tasks.

Note: A task can use SRP mutexes with other mutexes with different protocol (for example, PI, PC, NOP mutexes). We don't know the behaviour of that choice, but, if you want to try, it works!

Files: `include/modules/srp.h - kernel/modules/srp.c`

Implementation hints: Notes about the implementation of SRP are inserted in the heading the source file `kernel/modules/srp.c`.

[5] i.e., it was created without passing a `SRP_RES_MODEL` to the `task_createn` primitive.

Chapter 6

File system Modules

6.1 BD_EDF

TBD

6.2 BD_PSCAN

TBD